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, Dec 2022

Abstract

This thesis presents a prototype implementation of the syntax and semantics for specifying the default values for fields of Haskell record types. The proposed change aims to have no impact on existing Haskell code and minimal syntactic intrusion when the proposed feature is enabled. The use of default values happens only when using the braced record construction syntax already present in Haskell. The syntax for defining default field values follows established conventions from other programming languages in which this feature is present. The addition of this feature will noticably improve the language's usability in large codebases.

Keywords

functional programming, Haskell, compilers, syntax, extension, default value, records, fields

Thesis domain (Socrates-Erasmus subject area codes)

11.3 Informatyka

Subject classification

D. Software
D.3 Programming Languages
D.3.3 Language Constructs and Features

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 [7] 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, as well as examples of problems they solve. For the sake of providing context to subsequent information, we provide a simple overview of the architecture of the GHC compiler. Then, we outline the syntactic changes applied to the grammar of Haskell, the semantic behavior of introduced features and the purely semantic changes to existing constructs. In all of these, we provide alternative considered solutions and the reasoning behind the final choices. We also explain the development process and decisions made when implementing the feature in GHC, the leading compiler of the Haskell language.

Chapter 1

Background

1.1. Overview of record syntax and field defaulting

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.

As an example, let's compare record syntax in C and Haskell representing a person's name and age.

```
typedef struct {
  char* name;
  int age;
} Person;
```

```
data Person
  = Person
  { name :: String
  , age :: Int
  }
```

In C, without this syntax, the only way to create compound data structures consisting of data of different types is manual pointer arithmetic:

```
// create a record with this syntax and set fields
Person person;
person.name = "Adam";
person.age = 21;

// one way to recreate the above on Linux without struct syntax
void* person = alloca(sizeof(char*) + sizeof(int));
*(char**)person = "Adam";
*(int*)(person + sizeof(char*)) = 21;
// the above would of course be extracted to accessor functions
// in hypothetical real-world code
```

In Haskell, the lack of named fields would necessitate boilerplate accessor functions as described in [23].

1.1.1. History of records and field defaulting in other languages

The earliest examples of programming language syntax for record types date back to the 1950s and 60s with their introduction in COBOL in 1959 [14]. Popular contemporary languages like Algol 68 [22] and FORTRAN 77 [2] adopted their own syntax soon thereafter. They did not provide any facilities for providing default values for structure fields. Early object-oriented languages either used constructors for this purpose (Smalltalk [6]), or nothing at all (Simula [3]). The same followed for most of languages developed in the 1970s and 80s: languages with either no default initialization of record types, like C and Pascal, or default initialization via constructor functions, like in C++. A notable exception is Common Lisp, which being released in 1985 can be counted as one of the earliest examples of records with dedicated syntax for default field values [17]. With the rise of popularity of programming and the coming of the Internet era, syntactic support for default values has suddenly changed from obscure to ubiquitous, as will be clear from the examples in the next section.

1.1.2. Modern examples in other languages

Examining 10 of the most popular general-purpose programming languages in 2022, most of them provide in their latest version some utility for specifying default field values, either through constructor functions or dedicated syntax. Chosen programming languages are: C++, Java, JavaScript, TypeScript, Python, C#, PHP, Go, Kotlin and Rust. The choice of languages was based on statistics from [16], after removing markup languages, query languages and scripting languages, as well as C which was already discussed above.

Of course, any programming language offers support for default values through creating an object and using it as a "default" object. This approach is not investigated, as it requires no help from the language to implement, unless there is a language-wide convention on the details on how the object is supposed to be created. Furthermore, this approach has other drawbacks discussed in section 1.2.2.

The below table illustrates support for field defaulting in the above languages.

	Dedicated syntax	Parameterless constructors
C++	Since C++11	Yes
Java	Yes	Yes
JavaScript	Since ES6	Since ES6, or as prototypes
TypeScript	Yes	Yes
Python	dataclasses only	Yes
C#	Yes	Yes
PHP	Since PHP5	Since PHP5
Go	No	No
Kotlin	Yes	Yes
Rust	No	Limited, through the Default trait

Table 1.1: Support for specifying default field values across popular languages.

Below are examples of each syntactic construct mentioned in the table. Note that, of course, languages supporting the parameterless constructor syntax allow not only for providing default values for all fields at once, but also for providing values of an arbitrary subset of fields through simply adding parameters to the constructor.

- C++: dedicated syntax (C++11 onwards) [9]

```
struct Person {
  std::string name = "Adam";
  int age = 21;
};
```

  The above values get assigned to fields if the fields are missing in the constructor's member initialization list (showcased in the next bullet).

- C++: default constructor [8]

```
struct Person {
  std::string name;
  int age;
  Person() : name("Adam"), age(21) {}
};
```

  This approach uses the constructor's member initialization list, which guarantees that the fields are initialized with the default values before entering the constructor body.

- Java: dedicated syntax

```
class Person {
  public String name = "Adam";
  public int age = 21;
}
```

  This approach assures that the default values are assigned before entering the body of any user-defined constructor.

- Java: parameterless constructor

```
class Person {
  public String name;
```

```
  public int age;
  public Person() {
    name = "Adam";
    age = 21;
  }
}
```

This approach is just manual assignment of values, the only syntactic help is the constructor itself.

- JavaScript: dedicated syntax (since ES6)

```
class Person {
  name = "Adam";
  age = 21;
}
```

- JavaScript: parameterless constructor (since ES6)

```
class Person {
  constructor() {
    this.name = "Adam";
    this.age = 21;
  }
}
```

- JavaScript: prototypes

```
function Person() {
  this.name = "Adam";
  this.age = 21;
}
```

This defines a constructor function for `Person`, with any required methods being later added to the function's prototype.

- TypeScript: dedicated syntax

```
class Person {
  name = "Adam";
  age = 21;
}
```

Note that the type of the fields is automatically deduced from the initializing expression.

- TypeScript: parameterless constructor

```
class Person {
  name: string;
  age: number;
  constructor() {
    this.name = "Adam";
    this.age = 21;
  }
}
```

This approach is just manual assignment of values, the only syntactic help is the constructor itself.

- Python: dataclasses

```
@dataclass
class Person:
  name: str = "Adam"
  age: int = 21
```

- Python: parameterless constructor

```
class Person:
  def __init__(self):
    self.name = "Adam"
    self.age = 21
```

- C#: dedicated syntax

```
class Person {
  public String Name = "Adam";
  public int Age = 21;
}
```

This approach assures that the default values are assigned before entering the body of any user-defined constructor.

- C#: parameterless constructor

```
class Person {
  public String Name;
  public int Age;
  public Person() {
    Name = "Adam";
    Age = 21;
  }
}
```

This approach is just manual assignment of values, the only syntactic help is the constructor itself.

- PHP: dedicated syntax (since PHP 5)

```
class Person {
  public $name = "Adam";
  public $age = 21;
}
```

- PHP: parameterless constructor (since PHP 5)

```
class Person {
  public $name;
  public $age;
  function __construct() {
    $this->name = "Adam";
```

```php
    $this->age = 21;
  }
}
```

Note that this is not actually custom syntax, constructors in PHP are regular methods with the name `__construct`.

- Kotlin: dedicated syntax and parameterless constructor

  ```kotlin
  class Person {
    val name = "Adam";
    val age = 21;
  }
  ```

  The body of the class is explicitly considered to be the primary constructor in Kotlin.

- Kotlin: parameterless (secondary) constructor

  ```kotlin
  class Person {
    val name: String;
    val age: Int;
    constructor() {
      name = "Adam";
      age = 21;
    }
  }
  ```

- Rust: Default trait

  ```rust
  struct Person {
    name: String;
    age: i32;
  }

  impl Default for Person {
    fn default() -> Person {
      Person {
        name: "Adam",
        age: 21,
      }
    }
  }
  ```

  This approach is limited by only being able to provide default values to all fields of the struct at once. It also is not an example of syntactic support, as all that is required is a trait. Because of that, use of this trait for default construction relies entirely on convention.

1.1.3. Conclusion

From the above considerations it is clear that the vast majority of modern languages offer a convenient syntax for providing default field values. Comparing that to the historical situation where it was not offered by any major language, it is clear that it has become a quality-of-life feature that today's programmers have come to expect from their language of choice. Thus, an addition of such a feature is likely to make teams and developers more eager to adopt Haskell as their language of choice.

1.2. Field defaulting in Haskell

1.2.1. Introduction

Despite record syntax's relative ubiquity in modern functional programming languages, most implementations of it (e.g. OCaml, F#, ReasonML and of course Haskell) 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. The only example of this syntax in a popular functional programming language seems to be Erlang, where records can be default-constructed and the fields without default values are undefined [5].

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 of the type can utilize the record update syntax to mimic the functionality offered by default field values.

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 field value unnecessarily cumbersome to access after the creation of the record. Unlike its imperative counterpart, this approach does not offer the programmer any automated systems that would detect the absence of a value that is necessary for the type. 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 Haskell, as well as other languages, it is also possible to create a record without specifying all of its fields. This way it is not necessary to provide the troublesome fields with any value, relying on the user to add it later. However, in that case, the fields are still automatically filled with bottom values. To make matters worse, such a creation creates compile-time warnings (with `-Wmissing-fields` enabled) when defining the default record, and no compile-time warnings when some of the bottoms are not replaced in a record update. Because of that, this solution is even worse than the previous one.

This issue is completely solved with the advent of default field value syntax: all fields that can't be easily defaulted, are just not given a default value, and with `-Wmissing-fields` we get compile-time warnings whenever any non-defaulted field is missing.

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 close location, 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.

By contrast, none of these problems are present when using default field values.

1.2.3. Direct use cases

Aside from the obvious functionality improvements, allowing default fields in records has positive impact on the overall Haskell ecosystem. Many large multi-language codebases, such as the giant monorepos used by large companies, use interface definition languages (IDLs) like the ones used in Apache Thrift or Google's Protobuf for interfacing between systems written in different languages. Such IDLs usually provide this functionality through a language-agnostic syntax for declaring the general layout of data structures and/or procedures. 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 implement workarounds before codebases written in that language are able to intercommunicate via IDLs.

Code migrations

A major potential real-world use case is helping with code migrations. As an example, consider a system with two codebases, *user* and *infra*. *infra* provides type definitions for configs, like this simplified example:

```
data Config
  = Config
  { name :: String
  , value :: SomeType
  }
```

Code in *user* will then construct values of this type, like so:

```
someVar :: Config
someVar = Config { name = "my config", value = myValue }
```

To ensure runtime safety of the system, constructing this value is protected via the combination of the `-Wmissing-fields` and `-Werror` compiler flags. In their presence, omitting a field creates a compile-time error, making sure that every field value is correctly in place. Since it is easy for a programmer to forget about a field, especially when there are many of them, this check measurably increases correctness of the system. However, when there is a need to add an additional field, problems with that approach arise. Even if the system was limited to just one codebase, a sufficiently large system cannot be easily migrated to use a new value in all the occurrences of the construction. With multiple codebases, the process complicates even more. Here's a step-by-step example of how that could look like, if one wanted to add a field `owner` of type `String`:

1. We cannot add the field directly without creating compile errors in *user* code. To work around that, we create a default value called, let's say, `defaultConfig` in the same module of *infra* code as the definition of the type.

   ```
   defaultConfig :: Config
   defaultConfig = Config { name = "default name", value = defaultSomeTypeValue }
   ```

2. This code needs to be released in order for `defaultConfig` to be accessible in *user* code.

3. Then, all constructors of `Config` in *user* can be modified to a record update of `defaultConfig`:

   ```
   someVar :: Config
   someVar = defaultConfig { name = "my config", value = myValue }
   ```

   This change silences the warning

4. This code, again, needs to be released before any further changes in *infra* are made.

5. With that setup, we can safely introduce a new field in *infra*, immediately modifying `defaultConfig` with a default value :

   ```
   data Config
     = Config
     { name :: String
     , value :: SomeType
     , owner :: String
     }

   defaultConfig :: Config
   defaultConfig
     = Config
     { name = "default name"
     , value = defaultSomeTypeValue
     , owner = "default owner"
     }
   ```

6. Yet again, a release is needed.

7. Now, all of the *user* code can revert to using constructors, as soon as the relevant code is adapted to make use of the new field.

   ```
   someVar :: Config
   someVar
     = Config
     { name = "my config"
     , value = myValue
     , owner = "my owner"
     }
   ```

8. This *user* code must again be released. This restores the protection given by `-Wmissing-fields`.

By contrast, with the addition of field default value syntax the above process could be shortened to just one *infra* release containing the new field with the default value:

```
data Config
  = Config
  { name :: String
  , value :: SomeType
  , owner = "default owner" :: String
  }
```

This way, `-Wmissing-fields` is working for all the other fields, and teams working on *user* code can add support for the new field at their own pace. Once this is done, the default value can be removed to ensure it's also protected by `-Wmissing-fields`.

Example of a big config

A good example of a large record type in a widely used Haskell codebase is the DynFlags structure in GHC. Its type definition is indeed massive, and so is the function used for constructing its default value. The definition is included in Appendix A, and the default value construction in Appendix B. This is because the former takes up 6 pages and the latter 4 pages. Investigating the default value for the type, a couple of notable observations can be made.

- It's a function taking a `Settings` value as an argument, as some fields of the default value rely on settings.
- Most of the other field values are either constants or global values.
- Three fields contain bottom values that panic on evaluation.

Clearly, the values that do not depend on the setting value are great candidates for being defaulted in the type definition itself. Only 8 fields actually depend on the setting value, so with the default value extension the function can be reduced to:

```
defaultDynFlags :: Settings -> DynFlags
defaultDynFlags mySettings =
-- See Note [Updating flag description in the User's Guide]
  DynFlags {
    backend = platformDefaultBackend (sTargetPlatform mySettings),
    ghcNameVersion = sGhcNameVersion mySettings,
    fileSettings = sFileSettings mySettings,
    toolSettings = sToolSettings mySettings,
    targetPlatform = sTargetPlatform mySettings,
    platformMisc = sPlatformMisc mySettings,
    rawSettings = sRawSettings mySettings,
    generalFlags = EnumSet.fromList (defaultFlags mySettings),
  }
```

The fields with bottom values illustrates another possible use case: explicitly deferring missing fields check to runtime. While generally it's considered beneficial to catch missing fields during compile-time, it's clear that in some cases this cannot be easily done because of technical debt. Sometimes it's enough to wrap the field inside a `Maybe`, but it is not without its downsides. Providing a panic value as a temporary placeholder allows for better error messages in case of a programming error, as well as silencing any `-Wmissing-fields` warnings. With default field values this can be done once, globally, instead of having to manually ensure it in every constructor.

Chapter 2

GHC architecture

This section contains an abridged overview of the architecture of the frontend of the GHC compiler to serve as background for implementation details provided later. The compilation process in the GHC compiler consists of 6 main phases [4], the first 4 of which are:

1. Parsing
2. Renaming
3. Typechecking
4. Desugaring

These are all the phases that deal with the Haskell AST, mostly unchanged from the parser. The last one, the desugarer, desugars the Haskell AST to the much smaller internal language Core, which is used for optimization. Before discussing the passes themselves, it's worth to describe how GHC deals with modification of the AST throughout the frontend of the compiler.

2.1. Trees that grow

Throughout the phases of the frontend, various data about the Haskell AST is generated (and other becomes no longer needed). In order to avoid data structure fields or constructors being unused during some phases or having multiple slightly different versions of the same AST type definitions, a mechanism was introduced in [11] which parameterized the entire syntax tree with a type argument representing the current phase of compilation. Three distinct phases are available: `GhcPs`, `GhcRn`, `GhcTc`, with the parser producing code parameterized by `GhcPs`, the renamer converting `GhcPs` to `GhcRn`, and the typechecker converting `GhcRn` to `GhcTc`. With that in mind, when creating a data structure, the developer writes all constructors and fields common to all phases as they normally would, with a minor exception. Every type `T` gets an extra constructor `XT` (eXtension to `T`), which allows for phase-specific constructors, and every constructor `CtorT` has as its first field set to type `XCtorT`, to allow for phase-specific constructor fields. Any type written in extension field or an extension constructor definition is actually a `type family`, which when parameterized by the phase, allows for a different `type instance` for each phase.

2.2. Compilation phases

2.2.1. Parsing

GHC does parsing of source code through a monolithic Happy grammar [1], although this wasn't initially the case [10]. The data structures emitted by the parser are often ambiguous as to which production was actually used to parse a given part of the source code. This is a result of a policy of "overparse, then filter out the bad cases" [19]. For example, in certain contexts both patterns and expressions are valid, so knowing which grammar derivation to use would require infinite lookahead. This is solved through a validation monad which allows parsing contexts to specify which constructs are possible, failing the parse on the rest. Another system of note are the source location and annotation types, which are ingrained into the types that form the Haskell source code AST. Any parser node has to extract the location and annotation data from its subnodes, and very carefully combine them together, as there is no automatic mechanism for detecting errors in code like this.

2.2.2. Renaming

The task of the renamer is to match each name usage in the code with the matching definition. Its code is intertwined with the typechecker code, and uses the same monad [20].

2.2.3. Typechecking

Typechecking a module happens concurrently (but not in parallel) with renaming, because of Template Haskell splices. The typechecker uses the `Type` type, which is used to type Core, instead of the `HsType` created by the parser [21]. This is because the typechecking process is quite complicated, and thus requires a data structure that is easy to manipulate.

2.2.4. Desugaring

The desugarer transforms the Haskell AST (with the `GhcTc` phase indicator) into an explicitly type variant of System FC called Core [18]. The Core language AST consists of very few constructors, and great care is taken to not modify this language when not necessary, so any changes that are little more than syntactic sugar should be implemented as part of the desugarer.

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 many others, looks like this: `int x = 1;`. As another example, the same is accomplished in TypeScript with the syntax `x: number = 1,`. Many other examples presented in 1.1.2 also follow this pattern.

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:

```
data NewRecordType = NewRecordTypeCtor { fieldName = expression }
```

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 Haddock, but also 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 potential loss in functionality seems to have negligible cost.

3.1.3. Multiple names in one line

Regular Haskelly syntax for field declarations allows for multiple fields of the same type to be written on one line:

```
data NewRecordType
  = NewRecordTypeCtor {
    fieldName1, fieldName2, fieldName3 :: FieldType
  }
```

It's not a widely used feature, especially given its counterpart in C/C++ is considered bad style. Its use is not particularly readable either: programmers usually visually judge the number of fields in a type by the number of lines. It also makes the code harder to document as it's not clear how individual fields should be documented, and the Haddock docs [15] do not take that syntax into account. With that in mind, complicating this syntax even further with default values does not seem desirable:

```haskell
data NewRecordType
  = NewRecordTypeCtor {
    fieldName1 = expr1, fieldName2 = expr2, fieldName3 = expr3 :: FieldType
  }
```

Furthermore, it could create ambiguity for the reader when only the last field is defaulted:

```haskell
data NewRecordType
  = NewRecordTypeCtor {
    fieldName1, fieldName2, fieldName3 = expr :: FieldType
  }
```

In this example, it is not immediately clear whether `expr` is the default value for all fields, or only `fieldName3`. For these reasons, this feature was not included in the proposed syntax.

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.

3.3. Summary

All in all, the new syntax is as follows: inside a record constructor definitions, any field declared in the form `name :: Type` can optionally be annotated with a default value: `name = <expr> :: Type`, where `<expr>` is an arbitrary expression (of the same type as the field). 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:

```
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:

```
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-⊥ 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 ⊥ value, both in the current version of Haskell and with the changes described in this thesis applied to GHC.

```
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 $\perp$. The example record created by the following code is equal to `RecordCtor3 { field1 = 5, field2 = 4 }`.

```
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 is only evaluated at most once, if it's needed (more details in 5.5). 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.

Chapter 5

Implementation

We implemented the proposed change to the language inside the leading Haskell compiler, GHC (Glorious Haskell Compiler) [7]. 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

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 (using the Trees that grow technique described in 2.1 and in detail in [11]). 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 `FieldLabels`. 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 `FieldLabels` for the given constructor with the occurrences provided in a given expression. Thus, to exclude defaulted fields, it is sufficient to filter out the `FieldLabels` 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 (⊥) 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.

5.5. Notes on global bindings

A notable side effect of the decision to implement default values as global bindings is that it implicitly performs the optimization described in [12] as the full laziness transformation. This naturally is expected to result in the same consequences and considerations as outlined therein:

- Programs can be made faster by not repeating work (to compute the value) and allocations (to store the value).
- When the default value "is already a value, or reduces to a value with a negligible amount of work", the only potential gain is in allocations.
- The values can potentially cause a space leak (e.g. when a default value is an infinite list and more elements get evaluated than will be needed in the future)
- They are harder to garbage-collect

How do these apply to the typical use case of default field values? In a reasonable program, one would expect default values to be more likely to need to be reused. Thus, making sure that their value isn't recomputed on each evaluation of a constructor that does not contain the field seems like a good approach. It also follows the principle of least confusion [13]: it's not immediately obvious to a user writing a constructor with a defaulted field how much work will be performed on construction, so the best approach is to minimize it.

Furthermore, it's common for default values to require little to no computation. For example, a major use case of default values is placeholders meant to be filled out later. Often programmers choose the simplest possible value for a given type as the default, e.g. `0 :: Int`. With this in mind, it seems unlikely for programmers to provide default values that could cause a noticable space leak, and most of the time one might reasonably expect a bunch of allocations to be omitted. Unfortunately, there is no way to verify these assumptions without developing an alternative implementation that doesn't make the default values global.

Chapter 6

Conclusion

This work proposed a custom syntax for default record fields in Haskell, detailed a proof-of-concept implementation and discussed the design choices and tradeoffs behind the syntax and semantics of the implementation.

From this work it is clear that an implementation of default record field values in Haskell is possible, usable, useful and hopefully soon implemented into the mainline GHC. The syntax is clear and intuitive and the behavior of programs that use it is predictable and efficient. Programs using the extension are simpler, more expressive and have the potential to be faster than ones written without the extension.

The next step is to submit a proposal for the syntax to be accepted as a language extension in the GHC compiler, to bring this feature to the widest possible audience. Further discussion with the GHC maintainers will be required to finalize the design choices and implementation details, and this work will serve as a good starting point for that discussion.

Appendix A

GHC DynFlags type definition

```
data DynFlags = DynFlags {
  ghcMode              :: GhcMode,
  ghcLink              :: GhcLink,
  backend              :: !Backend,
   -- ^ The backend to use (if any).
   --
   -- Whenever you change the backend, also make sure to set 'ghcLink' to
   -- something sensible.
   --
   -- 'NoBackend' can be used to avoid generating any output, however, note that:
   --
   --  * If a program uses Template Haskell the typechecker may need to run code
   --    from an imported module. To facilitate this, code generation is enabled
   --    for modules imported by modules that use template haskell, using the
   --    default backend for the platform.
   --    See Note [-fno-code mode].


  -- formerly Settings
  ghcNameVersion   :: {-# UNPACK #-} !GhcNameVersion,
  fileSettings     :: {-# UNPACK #-} !FileSettings,
  targetPlatform   :: Platform,     -- Filled in by SysTools
  toolSettings     :: {-# UNPACK #-} !ToolSettings,
  platformMisc     :: {-# UNPACK #-} !PlatformMisc,
  rawSettings      :: [(String, String)],
  tmpDir           :: TempDir,

  llvmOptLevel         :: Int,        -- ^ LLVM optimisation level
  verbosity            :: Int,        -- ^ Verbosity level: see Note [Verbosity levels]
  debugLevel           :: Int,        -- ^ How much debug information to produce
  simplPhases          :: Int,        -- ^ Number of simplifier phases
  maxSimplIterations   :: Int,        -- ^ Max simplifier iterations
  ruleCheck            :: Maybe String,
  strictnessBefore     :: [Int],      -- ^ Additional demand analysis

  parMakeCount         :: Maybe Int,  -- ^ The number of modules to compile in parallel
                                      --   in --make mode, where Nothing ==> compile as
                                      --   many in parallel as there are CPUs.

```

```
  enableTimeStats     :: Bool,       -- ^ Enable RTS timing statistics?
  ghcHeapSize         :: Maybe Int, -- ^ The heap size to set.

  maxRelevantBinds    :: Maybe Int, -- ^ Maximum number of bindings from the type envt
                                    --   to show in type error messages
  maxValidHoleFits    :: Maybe Int, -- ^ Maximum number of hole fits to show
                                    --   in typed hole error messages
  maxRefHoleFits      :: Maybe Int, -- ^ Maximum number of refinement hole
                                    --   fits to show in typed hole error
                                    --   messages
  refLevelHoleFits    :: Maybe Int, -- ^ Maximum level of refinement for
                                    --   refinement hole fits in typed hole
                                    --   error messages
  maxUncoveredPatterns :: Int,      -- ^ Maximum number of unmatched patterns to show
                                    --   in non-exhaustiveness warnings
  maxPmCheckModels    :: Int,       -- ^ Soft limit on the number of models
                                    --   the pattern match checker checks
                                    --   a pattern against. A safe guard
                                    --   against exponential blow-up.
  simplTickFactor     :: Int,       -- ^ Multiplier for simplifier ticks
  dmdUnboxWidth       :: !Int,      -- ^ Whether DmdAnal should optimistically put an
                                    --   Unboxed demand on returned products with at most
                                    --   this number of fields
  specConstrThreshold :: Maybe Int, -- ^ Threshold for SpecConstr
  specConstrCount     :: Maybe Int, -- ^ Max number of specialisations for any one function
  specConstrRecursive :: Int,       -- ^ Max number of specialisations for recursive types
                                    --   Not optional; otherwise ForceSpecConstr can diverge.
  binBlobThreshold    :: Maybe Word, -- ^ Binary literals (e.g. strings) whose size is above
                                    --   this threshold will be dumped in a binary file
                                    --   by the assembler code generator. 0 and Nothing disables
                                    --   this feature. See 'GHC.StgToCmm.Config'.
  liberateCaseThreshold :: Maybe Int, -- ^ Threshold for LiberateCase
  floatLamArgs        :: Maybe Int, -- ^ Arg count for lambda floating
                                    --   See 'GHC.Core.Opt.Monad.FloatOutSwitches'

  liftLamsRecArgs     :: Maybe Int, -- ^ Maximum number of arguments after lambda lifting a
                                    --   recursive function.
  liftLamsNonRecArgs  :: Maybe Int, -- ^ Maximum number of arguments after lambda lifting a
                                    --   non-recursive function.
  liftLamsKnown       :: Bool,      -- ^ Lambda lift even when this turns a known call
                                    --   into an unknown call.

  cmmProcAlignment    :: Maybe Int, -- ^ Align Cmm functions at this boundary or use default.

  historySize         :: Int,       -- ^ Simplification history size

  importPaths         :: [FilePath],
  mainModuleNameIs    :: ModuleName,
  mainFunIs           :: Maybe String,
  reductionDepth      :: IntWithInf, -- ^ Typechecker maximum stack depth
  solverIterations    :: IntWithInf, -- ^ Number of iterations in the constraints solver
                                     --   Typically only 1 is needed

  homeUnitId_           :: UnitId,               -- ^ Target home unit-id
  homeUnitInstanceOf_   :: Maybe UnitId,         -- ^ Id of the unit to instantiate
```

```
    homeUnitInstantiations_ :: [(ModuleName, Module)], -- ^ Module instantiations

    -- Note [Filepaths and Multiple Home Units]
    workingDirectory    :: Maybe FilePath,
    thisPackageName     :: Maybe String, -- ^ What the package is called, use with multiple home u
    hiddenModules       :: Set.Set ModuleName,
    reexportedModules   :: Set.Set ModuleName,

    -- ways
    targetWays_         :: Ways,       -- ^ Target way flags from the command line

    -- For object splitting
    splitInfo           :: Maybe (String,Int),

    -- paths etc.
    objectDir           :: Maybe String,
    dylibInstallName    :: Maybe String,
    hiDir               :: Maybe String,
    hieDir              :: Maybe String,
    stubDir             :: Maybe String,
    dumpDir             :: Maybe String,

    objectSuf_          :: String,
    hcSuf               :: String,
    hiSuf_              :: String,
    hieSuf              :: String,

    dynObjectSuf_       :: String,
    dynHiSuf_           :: String,

    outputFile_         :: Maybe String,
    dynOutputFile_      :: Maybe String,
    outputHi            :: Maybe String,
    dynOutputHi         :: Maybe String,
    dynLibLoader        :: DynLibLoader,

    dynamicNow          :: !Bool, -- ^ Indicate if we are now generating dynamic output
                                  -- because of -dynamic-too. This predicate is
                                  -- used to query the appropriate fields
                                  -- (outputFile/dynOutputFile, ways, etc.)

    -- | This defaults to 'non-module'. It can be set by
    -- 'GHC.Driver.Pipeline.setDumpPrefix' or 'ghc.GHCi.UI.runStmt' based on
    -- where its output is going.
    dumpPrefix          :: FilePath,

    -- | Override the 'dumpPrefix' set by 'GHC.Driver.Pipeline.setDumpPrefix'
    --    or 'ghc.GHCi.UI.runStmt'.
    --    Set by @-ddump-file-prefix@
    dumpPrefixForce     :: Maybe FilePath,

    ldInputs            :: [Option],

    includePaths        :: IncludeSpecs,
    libraryPaths        :: [String],
```

```
  frameworkPaths      :: [String],  -- used on darwin only
  cmdlineFrameworks   :: [String],  -- ditto

  rtsOpts             :: Maybe String,
  rtsOptsEnabled      :: RtsOptsEnabled,
  rtsOptsSuggestions  :: Bool,

  hpcDir              :: String,    -- ^ Path to store the .mix files

  -- Plugins
  pluginModNames      :: [ModuleName],
    -- ^ the @-fplugin@ flags given on the command line, in *reverse*
    -- order that they're specified on the command line.
  pluginModNameOpts   :: [(ModuleName,String)],
  frontendPluginOpts  :: [String],
    -- ^ the @-ffrontend-opt@ flags given on the command line, in *reverse*
    -- order that they're specified on the command line.

  externalPluginSpecs :: [ExternalPluginSpec],
    -- ^ External plugins loaded from shared libraries

  --  For ghc -M
  depMakefile         :: FilePath,
  depIncludePkgDeps   :: Bool,
  depIncludeCppDeps   :: Bool,
  depExcludeMods      :: [ModuleName],
  depSuffixes         :: [String],

  --  Package flags
  packageDBFlags      :: [PackageDBFlag],
        -- ^ The @-package-db@ flags given on the command line, In
        -- *reverse* order that they're specified on the command line.
        -- This is intended to be applied with the list of "initial"
        -- package databases derived from @GHC_PACKAGE_PATH@; see
        -- 'getUnitDbRefs'.

  ignorePackageFlags  :: [IgnorePackageFlag],
        -- ^ The @-ignore-package@ flags from the command line.
        -- In *reverse* order that they're specified on the command line.
  packageFlags        :: [PackageFlag],
        -- ^ The @-package@ and @-hide-package@ flags from the command-line.
        -- In *reverse* order that they're specified on the command line.
  pluginPackageFlags  :: [PackageFlag],
        -- ^ The @-plugin-package-id@ flags from command line.
        -- In *reverse* order that they're specified on the command line.
  trustFlags          :: [TrustFlag],
        -- ^ The @-trust@ and @-distrust@ flags.
        -- In *reverse* order that they're specified on the command line.
  packageEnv          :: Maybe FilePath,
        -- ^ Filepath to the package environment file (if overriding default)


  -- hsc dynamic flags
  dumpFlags           :: EnumSet DumpFlag,
  generalFlags        :: EnumSet GeneralFlag,
```

```
  warningFlags        :: EnumSet WarningFlag,
  fatalWarningFlags   :: EnumSet WarningFlag,
  -- Don't change this without updating extensionFlags:
  language            :: Maybe Language,
  -- | Safe Haskell mode
  safeHaskell         :: SafeHaskellMode,
  safeInfer           :: Bool,
  safeInferred        :: Bool,
  -- We store the location of where some extension and flags were turned on so
  -- we can produce accurate error messages when Safe Haskell fails due to
  -- them.
  thOnLoc             :: SrcSpan,
  newDerivOnLoc       :: SrcSpan,
  deriveViaOnLoc      :: SrcSpan,
  overlapInstLoc      :: SrcSpan,
  incoherentOnLoc     :: SrcSpan,
  pkgTrustOnLoc       :: SrcSpan,
  warnSafeOnLoc       :: SrcSpan,
  warnUnsafeOnLoc     :: SrcSpan,
  trustworthyOnLoc    :: SrcSpan,
  -- Don't change this without updating extensionFlags:
  -- Here we collect the settings of the language extensions
  -- from the command line, the ghci config file and
  -- from interactive :set / :seti commands.
  extensions          :: [OnOff LangExt.Extension],
  -- extensionFlags should always be equal to
  --     flattenExtensionFlags language extensions
  -- LangExt.Extension is defined in libraries/ghc-boot so that it can be used
  -- by template-haskell
  extensionFlags      :: EnumSet LangExt.Extension,

  -- | Unfolding control
  -- See Note [Discounts and thresholds] in GHC.Core.Unfold
  unfoldingOpts       :: !UnfoldingOpts,

  maxWorkerArgs       :: Int,

  ghciHistSize        :: Int,

  flushOut            :: FlushOut,

  ghcVersionFile      :: Maybe FilePath,
  haddockOptions      :: Maybe String,

  -- | GHCi scripts specified by -ghci-script, in reverse order
  ghciScripts         :: [String],

  -- Output style options
  pprUserLength       :: Int,
  pprCols             :: Int,

  useUnicode          :: Bool,
  useColor            :: OverridingBool,
  canUseColor         :: Bool,
  colScheme           :: Col.Scheme,
```

```haskell

  -- | what kind of {-# SCC #-} to add automatically
  profAuto              :: ProfAuto,
  callerCcFilters       :: [CallerCcFilter],

  interactivePrint      :: Maybe String,

  -- | Machine dependent flags (-m\<blah> stuff)
  sseVersion            :: Maybe SseVersion,
  bmiVersion            :: Maybe BmiVersion,
  avx                   :: Bool,
  avx2                  :: Bool,
  avx512cd              :: Bool, -- Enable AVX-512 Conflict Detection Instructions.
  avx512er              :: Bool, -- Enable AVX-512 Exponential and Reciprocal Instructions.
  avx512f               :: Bool, -- Enable AVX-512 instructions.
  avx512pf              :: Bool, -- Enable AVX-512 PreFetch Instructions.

  -- | Run-time linker information (what options we need, etc.)
  rtldInfo              :: IORef (Maybe LinkerInfo),

  -- | Run-time C compiler information
  rtccInfo              :: IORef (Maybe CompilerInfo),

  -- | Run-time assembler information
  rtasmInfo              :: IORef (Maybe CompilerInfo),

  -- Constants used to control the amount of optimization done.

  -- | Max size, in bytes, of inline array allocations.
  maxInlineAllocSize  :: Int,

  -- | Only inline memcpy if it generates no more than this many
  -- pseudo (roughly: Cmm) instructions.
  maxInlineMemcpyInsns :: Int,

  -- | Only inline memset if it generates no more than this many
  -- pseudo (roughly: Cmm) instructions.
  maxInlineMemsetInsns :: Int,

  -- | Reverse the order of error messages in GHC/GHCi
  reverseErrors         :: Bool,

  -- | Limit the maximum number of errors to show
  maxErrors             :: Maybe Int,

  -- | Unique supply configuration for testing build determinism
  initialUnique         :: Word,
  uniqueIncrement       :: Int,
    -- 'Int' because it can be used to test uniques in decreasing order.

  -- | Temporary: CFG Edge weights for fast iterations
  cfgWeights            :: Weights
}
```

Appendix B

GHC DynFlags default value

```
-- | The normal 'DynFlags'. Note that they are not suitable for use in this form
-- and must be fully initialized by 'GHC.runGhc' first.
defaultDynFlags :: Settings -> DynFlags
defaultDynFlags mySettings =
-- See Note [Updating flag description in the User's Guide]
     DynFlags {
        ghcMode                 = CompManager,
        ghcLink                 = LinkBinary,
        backend                 = platformDefaultBackend (sTargetPlatform mySettings),
        verbosity               = 0,
        debugLevel              = 0,
        simplPhases             = 2,
        maxSimplIterations      = 4,
        ruleCheck               = Nothing,
        binBlobThreshold        = Just 500000, -- 500K is a good default (see #16190)
        maxRelevantBinds        = Just 6,
        maxValidHoleFits = Just 6,
        maxRefHoleFits   = Just 6,
        refLevelHoleFits = Nothing,
        maxUncoveredPatterns    = 4,
        maxPmCheckModels        = 30,
        simplTickFactor         = 100,
        dmdUnboxWidth           = 3,      -- Default: Assume an unboxed demand on function bodies re
        specConstrThreshold     = Just 2000,
        specConstrCount         = Just 3,
        specConstrRecursive     = 3,
        liberateCaseThreshold = Just 2000,
        floatLamArgs            = Just 0, -- Default: float only if no fvs
        liftLamsRecArgs         = Just 5, -- Default: the number of available argument hardware reg
        liftLamsNonRecArgs      = Just 5, -- Default: the number of available argument hardware reg
        liftLamsKnown           = False,  -- Default: don't turn known calls into unknown ones
        cmmProcAlignment        = Nothing,

        historySize             = 20,
        strictnessBefore        = [],

        parMakeCount            = Just 1,

        enableTimeStats         = False,
```

```
        ghcHeapSize           = Nothing,

        importPaths           = ["."],
        mainModuleNameIs      = mAIN_NAME,
        mainFunIs             = Nothing,
        reductionDepth        = treatZeroAsInf mAX_REDUCTION_DEPTH,
        solverIterations      = treatZeroAsInf mAX_SOLVER_ITERATIONS,

        homeUnitId_           = mainUnitId,
        homeUnitInstanceOf_   = Nothing,
        homeUnitInstantiations_ = [],

        workingDirectory      = Nothing,
        thisPackageName       = Nothing,
        hiddenModules         = Set.empty,
        reexportedModules     = Set.empty,

        objectDir             = Nothing,
        dylibInstallName      = Nothing,
        hiDir                 = Nothing,
        hieDir                = Nothing,
        stubDir               = Nothing,
        dumpDir               = Nothing,

        objectSuf_            = phaseInputExt StopLn,
        hcSuf                 = phaseInputExt HCc,
        hiSuf_                = "hi",
        hieSuf                = "hie",

        dynObjectSuf_         = "dyn_" ++ phaseInputExt StopLn,
        dynHiSuf_             = "dyn_hi",
        dynamicNow            = False,

        pluginModNames        = [],
        pluginModNameOpts     = [],
        frontendPluginOpts    = [],

        externalPluginSpecs   = [],

        outputFile_           = Nothing,
        dynOutputFile_        = Nothing,
        outputHi              = Nothing,
        dynOutputHi           = Nothing,
        dynLibLoader          = SystemDependent,
        dumpPrefix            = "non-module.",
        dumpPrefixForce       = Nothing,
        ldInputs              = [],
        includePaths          = IncludeSpecs [] [] [],
        libraryPaths          = [],
        frameworkPaths        = [],
        cmdlineFrameworks     = [],
        rtsOpts               = Nothing,
        rtsOptsEnabled        = RtsOptsSafeOnly,
        rtsOptsSuggestions    = True,
```

```
        hpcDir                = ".hpc",

        packageDBFlags        = [],
        packageFlags          = [],
        pluginPackageFlags    = [],
        ignorePackageFlags    = [],
        trustFlags            = [],
        packageEnv            = Nothing,
        targetWays_           = Set.empty,
        splitInfo             = Nothing,

        ghcNameVersion = sGhcNameVersion mySettings,
        fileSettings = sFileSettings mySettings,
        toolSettings = sToolSettings mySettings,
        targetPlatform = sTargetPlatform mySettings,
        platformMisc = sPlatformMisc mySettings,
        rawSettings = sRawSettings mySettings,

        tmpDir                = panic "defaultDynFlags: uninitialized tmpDir",

        llvmOptLevel          = 0,

        -- ghc -M values
        depMakefile       = "Makefile",
        depIncludePkgDeps = False,
        depIncludeCppDeps = False,
        depExcludeMods    = [],
        depSuffixes       = [],
        -- end of ghc -M values
        ghcVersionFile = Nothing,
        haddockOptions = Nothing,
        dumpFlags = EnumSet.empty,
        generalFlags = EnumSet.fromList (defaultFlags mySettings),
        warningFlags = EnumSet.fromList standardWarnings,
        fatalWarningFlags = EnumSet.empty,
        ghciScripts = [],
        language = Nothing,
        safeHaskell = Sf_None,
        safeInfer  = True,
        safeInferred = True,
        thOnLoc = noSrcSpan,
        newDerivOnLoc = noSrcSpan,
        deriveViaOnLoc = noSrcSpan,
        overlapInstLoc = noSrcSpan,
        incoherentOnLoc = noSrcSpan,
        pkgTrustOnLoc = noSrcSpan,
        warnSafeOnLoc = noSrcSpan,
        warnUnsafeOnLoc = noSrcSpan,
        trustworthyOnLoc = noSrcSpan,
        extensions = [],
        extensionFlags = flattenExtensionFlags Nothing [],

        unfoldingOpts = defaultUnfoldingOpts,
        maxWorkerArgs = 10,

```

```
          ghciHistSize = 50, -- keep a log of length 50 by default

          flushOut = defaultFlushOut,
          pprUserLength = 5,
          pprCols = 100,
          useUnicode = False,
          useColor = Auto,
          canUseColor = False,
          colScheme = Col.defaultScheme,
          profAuto = NoProfAuto,
          callerCcFilters = [],
          interactivePrint = Nothing,
          sseVersion = Nothing,
          bmiVersion = Nothing,
          avx = False,
          avx2 = False,
          avx512cd = False,
          avx512er = False,
          avx512f = False,
          avx512pf = False,
          rtldInfo = panic "defaultDynFlags: no rtldInfo",
          rtccInfo = panic "defaultDynFlags: no rtccInfo",
          rtasmInfo = panic "defaultDynFlags: no rtasmInfo",

          maxInlineAllocSize = 128,
          maxInlineMemcpyInsns = 32,
          maxInlineMemsetInsns = 32,

          initialUnique = 0,
          uniqueIncrement = 1,

          reverseErrors = False,
          maxErrors     = Nothing,
          cfgWeights    = defaultWeights
        }
```

Bibliography

[1] Andy Gill, S. M. Happy, a parser generator for haskell. https://www.haskell.org/happy/. Accessed: 2022-11-20.

[2] Brainerd, W. Fortran 77. *Commun. ACM 21*, 10 (oct 1978).

[3] Dahl, O.-J. *SIMULA 67 Common Base Language, (Norwegian Computing Center. Publication)*. 1968.

[4] Diehl, S. Dive into ghc: Intermediate forms. https://www.stephendiehl.com/posts/ghc_02.html, 2016. Accessed: 2022-11-12.

[5] Ericsson AB. Erlang programming examples user's guide - records. https://www.erlang.org/doc/programming_examples/records.html. Accessed: 2022-12-03.

[6] Goldberg, A., and Robson, D. *Smalltalk-80: The Language and Its Implementation*. Addison-Wesley Longman Publishing Co., Inc., USA, 1983.

[7] Hudak, P., Peyton Jones, S., Wadler, P., Boutel, B., Fairbairn, J., Fasel, J., Guzmán, M., Hammond, K., Hughes, J., Johnsson, T., Kieburtz, R., Nikhil, R., Partain, W., and Peterson, J. Report on the programming language haskell: a non-strict, purely functional language version 1.2. *SIGPLAN Notices 27* (01 1992), 1–.

[8] ISO. *ISO/IEC 14882:1998: Programming languages — C++*. pub-ISO, pub-ISO:adr, Sept. 1998. Available in electronic form for online purchase at http://webstore.ansi.org/ and http://www.cssinfo.com/.

[9] ISO. *ISO/IEC 14882:2011 Information technology — Programming languages — C++*, third ed. pub-ISO, pub-ISO:adr, Sept. 2011.

[10] Jones, S. P., Hammond, K., Partain, W., Wadler, P., Hall, C. B., and Jones, S. L. P. The glasgow haskell compiler: a technical overview.

[11] Najd, S., and Jones, S. L. P. Trees that grow. *ArXiv abs/1610.04799* (2017).

[12] Partain, W., Santos, A., and Peyton Jones, S. Let-floating: moving bindings to give faster programs, May 1996. ACM SIGPLAN International Conference on Functional Programming (ICFP'96).

[13] Saltzer, J., and Kaashoek, F. *Principles of Computer System Design: An Introduction*. Morgan Kaufmann, 2009.

[14] Sebesta, R. *Concepts of Programming Languages*. Benjamin/Cummings series in computer science. Addison-Wesley Publishing Company, 1996.

[15] Simon Marlow. Haddock docs - documenting a top-level declaration. https://haskell-haddock.readthedocs.io/en/v2.20/markup.html. Accessed: 2022-12-04.

[16] Statista. Most used programming languages among developers worldwide as of 2022. https://www.statista.com/statistics/793628/worldwide-developer-survey-most-used-languages/. Accessed: 2022-11-25.

[17] Steele, G. L. *Common LISP: The Language*. Digital Press, USA, 1984.

[18] The Glasgow Haskell Team. Ghc wiki commentary: The core language. https://gitlab.haskell.org/ghc/ghc/-/wikis/commentary/compiler/core-syn-type. Accessed: 2022-11-21.

[19] The Glasgow Haskell Team. Ghc wiki commentary: The parser. https://gitlab.haskell.org/ghc/ghc/-/wikis/commentary/compiler/parser. Accessed: 2022-11-20.

[20] The Glasgow Haskell Team. Ghc wiki commentary: The renamer. https://gitlab.haskell.org/ghc/ghc/-/wikis/commentary/compiler/renamer. Accessed: 2022-11-21.

[21] The Glasgow Haskell Team. Ghc wiki commentary: The typechecker. https://gitlab.haskell.org/ghc/ghc/-/wikis/commentary/compiler/type-checker. Accessed: 2022-11-21.

[22] van Wijngaarden, A., Mailloux, B., Peck, J., Koster, C., Lindsey, C., Sintzoff, M., Meertens, L., and Fisker, R. *Revised Report on the Algorithmic Language Algol 68*. Springer Berlin Heidelberg, 2012.

[23] Wikibooks. Haskell/more on datatypes. https://en.wikibooks.org/w/index.php?title=Haskell/More_on_datatypes&oldid=3676091. Accessed: 2022-11-23.