Karlsruher Institut für Technologie

Institut für Programmstrukturen
und Datenorganisation (IPD)

Lehrstuhl Prof. Dr.-Ing. Snelting

# Structural Diffing for GHC Core Programs

Bachelorarbeit von

**Paul Brinkmeier**

an der Fakultät für Informatik

```
$trModule_s1vV and $trModule_sTY are equivalent
$trModule_s1vT and $trModule_sTW are equivalent
$trModule_s1vW and $trModule_sTZ are equivalent
$trModule_s1vU and $trModule_sTX are equivalent
lvl_s1w3 and lvl_sU6 are equivalent
lvl_s1w5 and lvl_sU8 are equivalent
lvl_s1w1 and lvl_sU4 are equivalent
$trModule_r1vh and $trModule_rTm are equivalent
--- fac_rjM
+++ fac_rjH
@@ -31,7 +31,7 @@
              in let [LclId,
                      Arity=1,
                      Occ=Strong Loopbrk,
-                     Str=<L,U> {a1vi-><S(C(C(S))L),U(C(C1(U)),A)>}]
+                     Str=<L,U> {aTn-><S(C(C(S))L),U(C(C1(U)),A)>}]
                    fac_s1w0 :: p_al7 -> p_al7
                    fac_s1w0 =
                      \ ds_d1vG ->
```

| | |
|---|---|
| **Erstgutachter:** | Prof. Dr.-Ing. Gregor Snelting |
| **Zweitgutachter:** | Prof. Dr. rer. nat. Bernhard Beckert |
| **Betreuende Mitarbeiter:** | M. Sc. Sebastian Graf |
| **Abgabedatum:** | 23. Oktober 2020 |

# Zusammenfassung

Der Glasgow Haskell Compiler (GHC) setzt zur Implementierung von Optimierungen die Zwischensprache Core ein. Während der Entwicklung solcher Optimierungen werden zur Fehlersuche Vergleiche zwischen Core-Programmen herangezogen, die von verschiedenen GHC-Versionen erzeugt wurden.

Diese Vergleiche werden textbasiert, beispielsweise per `diff`, durchgeführt. Auf diese Weise werden viele falsch-positive Unterschiede gefunden. Da verschiedene GHC-Versionen unterschiedliche Variablennamen erzeugen, ist es beispielsweise so nicht immer möglich, $\alpha$-Äquivalenz von Programmen festzustellen.

Diese Arbeit stellt ein Vergleichswerkzeug vor, das Core-Programme nicht textbasiert sondern in ihrer Struktur betrachtet. Dieser Ansatz ermöglicht es, bestimmte irrelevante Unterschiede auszublenden.

The Glasgow Haskell Compiler (GHC) uses its intermediate language Core to implement optimization passes. During the development of such optimization passes, Core programs generated by different versions of GHC are compared for debugging purposes.

These comparisons are drawn textually, using for example the utility `diff`. In this manner, many false positive differences are found. Because different versions of GHC generate different names for variables, it is for example not always possible to determine $\alpha$-equivalence in this way.

This thesis presents a comparison utility, which compares Core programs not textually but in their structure. This approach makes it possible to hide some irrelevant differences.

# Contents

# 1. Introduction

Functional programming (FP) languages such as Haskell pioneered language features that recently made their way into mainstream programming languages. For example, lambda functions and local variable type inference, both a part of Haskell since its inception in 1990, were implemented in Java 8 (2014) and Java 10 (2018) respectively. This makes Haskell a suitable staging area for a variety of language and compiler features.

The Glasgow Haskell Compiler (GHC) manipulates a variety of intermediate representations during compilation. Dividing compilation into multiple passes allows the GHC development community to independently work on compiler features on different levels of abstraction. One of these intermediate representations is GHC Core [1], which is used to implement high-level optimization passes that, among other things, leverage the language's strict type system and FP features such as laziness.

In order to debug optimization passes in Core, GHC developers compare the outputs of different compiler runs. For this purpose, they compile the same file using two different versions of GHC and inspect the generated Core programs.

A widespread and simple to use tool for comparing computer code is `diff`, which compares two files line by line and displays the ones that are different.

In some cases, `diff` by itself is insufficient for comparing Core programs. For example, consider the following snippets of Core programs generated from the same Haskell program by different versions of GHC. The first version generates:

```
a'_sNB
  = \ (s1_aN5 :: Int) ->
      case s1_aN5 of { GHC.Types.I# x_aNl ->
      a'_sNB (GHC.Types.I# (GHC.Prim.+# x_aNl 1#))
      }
```

The second version generates:

```
a'_sLj
  = \ (eta2_aKN :: Int) ->
      case eta2_aKN of { GHC.Types.I# x_aL3 ->
      a'_sLj (GHC.Types.I# (GHC.Prim.+# x_aL3 1#))
      }
```

Looking closely, we notice that `a'_sNB` and `a'_sLj` are the same function. They are in fact "$\alpha$-equivalent", i.e. they share the same structure and differ only in the

names of their variables. `diff`, comparing only the textual form of the functions, does not recognize this:

```
-a'_sNB
-  = \ (s1_aN5 :: Int) ->
-      case s1_aN5 of { GHC.Types.I# x_aNl ->
-      a'_sNB (GHC.Types.I# (GHC.Prim.+# x_aNl 1#))
+a'_sLj
+  = \ (eta2_aKN :: Int) ->
+      case eta2_aKN of { GHC.Types.I# x_aL3 ->
+      a'_sLj (GHC.Types.I# (GHC.Prim.+# x_aL3 1#))
       }
```

The foremost reason for these differences is the "unique" identifier GHC attaches to each variable name. The generation of these tags is not uniform across GHC versions. This leads to a high number of false positives when comparing programs, even if they are structurally equal.

In this thesis, we will develop CoreDiff, an utility for comparing GHC Core programs. Unlike `diff`, which draws a purely textural comparison, we take a structural approach. In order to reduce the output of irrelevant differences, we will pair up definitions from two Core programs and assimilate them using nominal techniques inspired by [2] before they are compared. Next to using `diff` for comparison, our utility also offers to use a tree-based differencing algorithm derived from the algorithm presented in [3]. This algorithm calculates differences not by inspecting the lines of the programs' textual representation, but their inherent tree structure.

In Chapter 2, we will elaborate on the example above in order to demonstrate the problems CoreDiff solves more specifically and give some background on the approaches we used to solve them. We will then give a general overview of how we applied these concepts in Chapter 3. Chapter 4 presents challenges encountered during implementation. In Chapter 5, we will show some exemplary output of CoreDiff and compare its results to `diff`. Chapter 6 concludes this thesis by discussing the practical use of CoreDiff and listing some opportunities to improve it.

# 2. Preliminaries

Before we get into the gritty technical details, we will illustrate the specific problems we are trying to solve with an example. For this example, we chose the module `T18231` from the GHC test suite:

```
module T18231 where

import Control.Monad (forever)
import Control.Monad.Trans.State.Strict

m :: State Int ()
m = forever $ modify' (+1)
```

This module is referenced in GHC merge request !4207 [1]. It defines a single variable `m`, which represents an infinite loop counting up in the `State` monad [2].

The changes made to arity analysis in this merge request lead to unexpected compilation behavior: Without these changes, `m` – while not having a visible effect – compiles and runs without problems. With these changes, running `m` makes the run time system panic because of an infinite loop. We can replicate this behavior by compiling the following module, which lets `m` count up from 0:

```
module Main where

import T18231
import Control.Monad.Trans.State.Strict

main = print $ runState m 0
```

We compile and run it using a GHC `master` checkout from June 2020 [3], which we will refer to as GHC 8.92de9e, according to the major version it reports for itself and the first letters of its commit hash:

```
$ ghc8.92de9e --version
The Glorious Glasgow Haskell Compilation System, version 8.11.0.20200610

$ ghc8.92de9e -O2 Main.hs -o t18231
```

[1] See `https://gitlab.haskell.org/ghc/ghc/-/merge_requests/4207#note_304817`

[2] From the `transformers` package: `https://hackage.haskell.org/package/transformers-0.5.6.2/docs/Control-Monad-Trans-State-Strict.html#t:State`

[3] Git commit hash: `92de9e25aa1a6f7aa73154868521bcf4f0dc9d1e`

```
[1 of 2] Compiling T18231           ( T18231.hs, T18231.o )
[2 of 2] Compiling Main             ( Main.hs, Main.o )
Linking t18231 ...

$ ./t18231
[No output, program "hangs"]
```

The program prints nothing and just "hangs" – which is exactly what it should do, since we have not programmed it to do anything but count up infinitely. Using a system monitoring tool we can see that the program is in fact doing *something*, as it consumes a whole CPU core. We will now use the GHC version from merge request !4207 [4] to compile and run *the same program*. We refer to this version as GHC 9.ef9dbc.

```
$ ghc9.ef9dbc --version
The Glorious Glasgow Haskell Compilation System, version 9.1.0.20201013

$ ghc9.ef9dbc -O2 Main.hs -o t18231
[1 of 2] Compiling T18231           ( T18231.hs, T18231.o )
[2 of 2] Compiling Main             ( Main.hs, Main.o )
Linking t18231 ...

$ ./t18231
t18231: <<loop>>

$ echo $?
1
```

The program immediately terminates with the message `<<loop>>` and an exit status of 1 [5], which indicates that the run time system detected an infinite loop.

In order to discover the reason for this behavior, we will inspect the differences in the optimizations the two GHC versions perform on the intermediate language Core. The canonical way to do this comparison is to have two versions of GHC print out the Core programs generated by each of their passes and compare them using a textual differencing tool like `diff` or `vimdiff`, e.g.:

```
$ ghc8.92de9e -O2 T18231.hs -dverbose-core2core > T18231.ghc8.verbose
$ ghc9.ef9dbc -O2 T18231.hs -dverbose-core2core > T18231.ghc9.verbose
# Flags are chosen for reproducibility.
$ diff -u --color=always T18231.ghc8.verbose T18231.ghc9.verbose
```

In the following sections, we will take an in-depth look at the results of this comparison and consider some approaches to improve it. We will begin by giving some background information on the Core language itself.

[4] Git commit hash: `ef9dbc8f403194d4422a551bf6874c6d532f92bd`

[5] YMMV on a non-Unix system.

## 2.1. The Core language

Haskell is a large programming language with over fifty constructors for terms. Consequently, GHC converts it into an intermediate language called Core which only has ten constructors. Core is capable of representing any Haskell program but uses a vastly smaller syntax. For example, `do`-blocks, list comprehensions and nested pattern matching are all syntactical features of Haskell not present in Core. In order to transform a Haskell abstract syntax tree (AST) into Core, it first needs to be type checked as each Core term has an explicit type. The resulting fully typed Haskell AST is then "de-sugared", i.e. converted into a Core AST. [4, 5, 6]

GHC implements a variety of optimizations and analysis passes as Core-to-Core passes. Optimizations are meant to increase the efficiency of a program, for example by eliminating unused variables. Analysis passes collect information about the programs and pass it on to succeeding passes. This additional information is used to make succeeding optimization passes more effective. [4]

An overview of the passes run by GHC 8.92de9e and GHC 9.ef9dbc is shown in Table 5.1.

**Notation** A grammar for a subset of the System $F_C$ language, which the theory behind Core is based on, is provided in Figure 2.1. System $F_C$ was presented by Sulzmann et al. in [1]. It is an extension of the second-order lambda calculus (see for example [7, Sec. 1]), which introduces polymorphism via type abstractions. The grammar we provide is derived from the syntax presented by Eisenberg in [8], in which he explains some aspects of how System $F_C$ is implemented in GHC. We will use this syntax to describe high-level concepts. We will use a monospaced font with an ASCII-based character set when we discuss details of source code, GHC output or CoreDiff output.

**Programs** We consider a Core program to simply be a list of bindings, even though bindings can also be grouped into recursive block. This simplifies some aspects of our implementation. Some of these bindings are marked as *exported*, i.e. importable from other modules. These usually coincide with user-defined functions and types. Exported binder have unambiguous *names*.

**Binders and variables** We refer to the elements of the set Var as *binders* when they occur in a binding site, such as a lambda function or a let expression and as *variables* when they occur as a variable occurrence.

## 2.2. Binder problems

Using the flag `-dverbose-core2core` above, we let GHC print out the result of all Core-to-Core passes. The first of interest to us is titled `Desugar (after optimization)`. Consider the following snippet from that pass' difference:

| | | | |
|---|---|---|---|
| $P \in \mathsf{Prog}$ | $::=$ | $B_1; ...; B_n$ | Program |
| $B \in \mathsf{Bind}$ | $::=$ | $b = e$ | Binding |
| $x, b, b_i \in \mathsf{Var}$ | | | Binder |
| $\mathbb{K} \in \mathsf{DataCon}$ | | | Data constructor |
| $\mathbb{T} \in \mathsf{TyCon}$ | | | Type constructor |
| $e, f, e_i \in \mathsf{Exp}$ | $::=$ | | Expression |
| | $\mid$ | $x$ | Variable occurrence |
| | $\mid$ | $\lambda b.\, e$ | Term abstraction |
| | $\mid$ | $\Lambda b.\, e$ | Type abstraction |
| | $\mid$ | $e\ e$ | Term application |
| | $\mid$ | $e\ @\tau$ | Type application |
| | $\mid$ | $\textbf{let}\ b = e\ \textbf{in}\ f$ | Let expression |
| | $\mid$ | $\textbf{letrec}\ \overline{b_i = e_i}^{\,i}\ \textbf{in}\ e$ | Recursive let expression |
| | $\mid$ | $\textbf{case}\ e\ \textbf{of}\ b\ \{\ \overline{p_i \to e_i}^{\,i}\ \}$ | Case expression |
| $p_i \in \mathsf{Pat}$ | $::=$ | | Case pattern |
| | $\mid$ | $\mathbb{K}\ \overline{b_i}^{\,i}$ | Data constructor |
| | $\mid$ | $\sqcup$ | Wildcard |
| $\tau, \tau_i \in \mathsf{Typ}$ | $::=$ | | Type |
| | $\mid$ | $b$ | Type variable occurrence |
| | $\mid$ | $\forall b.\, \tau$ | Type polymorphism |
| | $\mid$ | $\tau \to \tau$ | Function type |
| | $\mid$ | $\tau\ \tau$ | Type application |
| | $\mid$ | $\mathbb{T}\ \overline{\tau_i}^{\,i}$ | Type constructor application |

**Figure 2.1.:** Expressions and types. Loosely follows the syntax provided in [1].

```
   (modify'
      @Data.Functor.Identity.Identity
      @Int
      Data.Functor.Identity.$fMonadIdentity
      (let {
-        ds_dM0 :: Int
+        ds_dJI :: Int
-        ds_dM0 = GHC.Types.I# 1# } in
-     \ (ds_dLZ :: Int) -> + @Int GHC.Num.$fNumInt ds_dLZ ds_dM0))
+        ds_dJI = GHC.Types.I# 1# } in
+     \ (ds_dJH :: Int) -> + @Int GHC.Num.$fNumInt ds_dJH ds_dJI))
```

Both versions of GHC essentially generate the same code. The content of the shown let expression is the same except for the binders's name. This happens because GHC assigns a special identifier to each binder, called its *unique*.

In some cases, names alone are not enough to disambiguate between binders. This is why each binder in Core is also given a "unique" identifier, simply referred to as its unique. Uniques are used for fast comparison of various data structures with an identity in GHC. The relation of a binder and its attached unique is only guaranteed to be valid within a single invocation of GHC. Uniques are never part of any compiler output except for debugging. [9]

We usually have to assume that two different versions of GHC assign different uniques to binders. This obscures actual differences in the programs.

### 2.2.1. The current solution

The GHC User's Guide describes the current solution to this problem in its documentation for the command-line flag `-dsuppress-uniques`:

> "Suppress the printing of uniques. This may make the printout ambiguous (e.g. unclear where an occurrence of 'x' is bound), but it makes the output of two compiler runs have many fewer gratuitous differences, so you can realistically apply `diff`. Once `diff` has shown you where to look, you can try again without `-dsuppress-uniques`." [10, Suppressing unwanted information]

This works for the example above, but the documentation already states that it is not completely satisfactory to simply suppress uniques completely. Consider this snippet from the difference of the next pass, the initial `Simplifier` run:

```
 [LclId,
  Arity=1,
+ Str=<L,U>b,
+ Cpr=b,
  Unf=...]
```

```
-a'_sNw
-  = \ (s1_aN0 :: Int) ->
-      case s1_aN0 of { GHC.Types.I# x_aNg ->
-      a'_sNw (GHC.Types.I# (GHC.Prim.+# x_aNg 1#))
+a'_sLe
+  = \ (eta2_aKI :: Int) ->
+      case eta2_aKI of { GHC.Types.I# x_aKY ->
+      a'_sLe (GHC.Types.I# (GHC.Prim.+# x_aKY 1#))
       }

-m = a'_sNw
+m = a'_sLe
```

Here, both versions of GHC have drastically simplified the body of `m` and moved it into a new binding `a'`. These differ not only in the variables' uniques, but also in their names: GHC 8.92de9e calls the parameter of the newly introduced abstraction term `s1`, GHC 9.ef9dbc calls it `eta2`. When we compare this snippet again using `-dsuppress-uniques`, two lines are still marked as different even though, for our purposes, they are not.

A difference that is actually important to us is that GHC 9.ef9dbc adds the fields `Str=<L,U>b` and `Cpr=b` to the metadata of `a'`, because they will influence the way succeeding optimization passes behave.

We would like our tool to mark the whole snippet, except for the metadata, as equivalent. Specifically, we are looking for a solution that makes terms equal where they are $\alpha$-equivalent. To achieve this, we consider two alternative approaches which we will outline in the next two sections.

### 2.2.2. De Bruijn index

The *namefree* form for lambda terms introduced by de Bruijn in [11] removes the need for variables to have names. In de Bruijn's namefree form, a variable occurrence specifies its binding site by a positive integer. This integer indicates the distance of an occurrence to its binding $\lambda$. Consider for example the $S$ combinator, which can be written in an infinite number of $\alpha$-equivalent ways:

$$\begin{aligned} S = \quad & \lambda x.\, \lambda y.\, \lambda z.\, x\ z\ (y\ z) \\ =_\alpha \quad & \lambda a.\, \lambda b.\, \lambda c.\, a\ c\ (b\ c) \\ =_\alpha \quad & \lambda s.\, \lambda k.\, \lambda i.\, s\ i\ (k\ i) \\ =_\alpha \quad & \ldots \end{aligned}$$

Its equivalent single namefree form is:

$$S = \lambda\, \lambda\, \lambda\, 3\ 1\ (2\ 1)$$

A conversion into namefree form hides any differences in binder names, i.e. makes $\alpha$-equivalent terms equal.

### 2.2.3. Nominal techniques

In [2], Gabbay and Pitts explain how the permutation model of set theory can be used to abstractly describe data types that support name abstraction modulo binder renaming. Their approach represents a straightforward alternative to higher-order abstract syntax (HOAS). Such "nominal" approaches are distinguished in that they use permutative variable renaming instead of substitution. For a $\lambda$-term $M$ and $a', a$ from the set of possible binders, they define:

> "$(a'\ a) \cdot M$, the *transposition* of *all* occurrences (be they free, bound, or binding) of $a$ and $a'$ in $M$." [2, Def. 2.1]

For example, $(a\ b) \cdot a.\, b = b.\, a$.

Urban et al. use this technique in [12] to define a "second-order" unification algorithm for terms involving binding operations. In [12, Fig. 2], they define a relation $\approx$, which can be used to test terms for $\alpha$-equivalence. For abstraction terms, $\approx$ is defined as such:

$$\frac{\nabla \vdash t \approx t'}{\nabla \vdash a.\, t \approx a.\, t'} \qquad \frac{a \neq a' \qquad \nabla \vdash t \approx (a\ a') \cdot t' \qquad \nabla \vdash a \# t'}{\nabla \vdash a.\, t \approx a'.\, t'}$$

The second rule reads: In the case that $a$ and $a'$ are not equal, the two abstraction terms $a.\, t$ and $a'.\, t'$ are $\alpha$-equivalent if you can swap *each* occurrence of $a$ and $a'$ in $t'$ to make it $\alpha$-equivalent to $t$ and $a$ does not occur freely in $t'$.

We used some of those concepts presented in [2] and [12] to develop a transformation that assimilates Core terms respective their binders by making them $\alpha$-equivalent where possible in Section 3.2.2.

## 2.3. Pairing program bindings

In order to avoid bindings overlapping in our comparison, we will first need to find pairs of corresponding bindings from our input programs. Consider this snippet from the difference of the first `Float out` passes:

```
-a'_sNw [Occ=LoopBreaker]
+lvl_sLm
+  =  (x_aKY :: GHC.Prim.Int#) ->
+       a'_sLe (GHC.Types.I# (GHC.Prim.+# x_aKY 1#))
+a'_sLe [Occ=LoopBreaker]
-a'_sNw
-  = \ (s1_aN0 :: Int) ->
-      case s1_aN0 of { GHC.Types.I# x_aNg ->
-      a'_sNw (GHC.Types.I# (GHC.Prim.+# x_aNg 1#))
-      }
+a'_sLe
+  = \ (eta2_aKI :: Int) ->
+      case eta2_aKI of { GHC.Types.I# x_aKY -> lvl_sLm x_aKY }
```

This is the first pass we considered so far where the two versions of GHC produce programs of different structure. We can observe that GHC 9.ef9dbc moved a part of `a'` into another binding `lvl`. GHC 8.92de9e does not apply this transformation.

Because we applied `diff` to the whole program, this difference is obscured by formatting artifacts: The first line, `a'_sNw [Occ=LoopBreaker]` is a part of `a'_sNw`'s signature and should therefore be listed next to its binding. We can reduce such confusing artifacts by applying `diff` to corresponding top level bindings instead of whole programs. We develop an algorithm that finds pairs of corresponding top level bindings in two Core programs in Section 3.1.

## 2.4. Differencing algorithms

So far, we have been using `diff` to compare Core programs. `diff` finds a minimal list of changes that transform one file into the other. We refer to this approach as *textual differencing*. Textual differencing was first considered from a theoretical viewpoint by Hunt and MacIlroy in [13]. Its approach is universally applicable across programming languages as it is based on the textual representation of code. Furthermore, it is intuitive for developers, since they are used to working with code in textual form.

### 2.4.1. Textual differencing

A downside of textual differencing is that it does not consider the underlying structure of the given programs. Reconsider the previous example:

```
-a'_sNw [Occ=LoopBreaker]
+lvl_sLm
+  =  (x_aKY :: GHC.Prim.Int#) ->
+       a'_sLe (GHC.Types.I# (GHC.Prim.+# x_aKY 1#))
+a'_sLe [Occ=LoopBreaker]
-a'_sNw
-  = \ (s1_aN0 :: Int) ->
-      case s1_aN0 of { GHC.Types.I# x_aNg ->
-      a'_sNw (GHC.Types.I# (GHC.Prim.+# x_aNg 1#))
-      }
+a'_sLe
+  = \ (eta2_aKI :: Int) ->
+      case eta2_aKI of { GHC.Types.I# x_aKY -> lvl_sLm x_aKY }
```

We have already discussed that the first line belongs with `a'_sNw`'s definition. We will solve this by recognizing the program structure by comparing corresponding bindings instead of whole programs.

Textual differencing does not recognize that certain differences stem purely from output formatting. In `a'_sNw`, the case expression's closing brace gets its own line. In `a'_sLe`, the whole case expression is printed on a single line. The actual difference

here only lies in the right-hand side of the single alternative appearing in this case expression.

### 2.4.2. Tree-based differencing

To avoid finding such non-differences, we consider the *tree-based* differencing algorithm presented by Miraldo and Swierstra in [3]. This algorithm does not generate a list of insertions and deletions as textual differencing algorithms do. Instead, it finds the "spine", i.e. the common constructors, of the left tree and the right tree. In places where the trees disagree, it leaves holes representing the changes by a pair of trees from the left and right tree respectively.

We will explain their algorithm with an example they give in [3, Sec. 2]. Their algorithm is data type generic, but for illustration their example uses a specific instance for 2-3-trees:

```
data Tree23 = Leaf
            | Node2 Tree23 Tree23
            | Node3 Tree23 Tree23 Tree23

data Tree23C h = LeafC
               | Node2C Tree23C Tree23C
               | Node3C Tree23C Tree23C Tree23C
               | Hole h
```

The data type `Tree23` represents a 2-3-tree. The data type `Tree23C h` is used to represent 2-3-trees with different kinds of holes of type `h`.

Our example trees are

```
x = Node2 r (Node2 s t)
y = Node2 r (Node2 t t)
```

where `r`, `s` and `t` are any 2-3-trees.

Their algorithm works in three steps:

**Computing changes** A change is a pair of `Tree23C Int`, where common subtrees of the left and right side are identified by integers. The left side of a change is called the deletion context; the right side is called the insertion context. In our case, the common subtrees of `x` and `y` are `r` and `t`:

```
change x y =
  ( Node2C (Hole 0) (Node2C s        (Hole 1))
  , Node2C (Hole 0) (Node2C (Hole 0) (Hole 1))
  )
```

This minimizes redundancy in the data structure representing the resulting patch. Given just `x` and this change, we can now already reconstruct `y`.

**Identifying the spine** The algorithm finds the constructors appearing in the same place on both sides of the change. This reduces redundancy further. For example, the two `Node2C` constructors are redundant in the change above. The spine is also called the *greatest common prefix* (GCP). In order to represent it, we use the type `Tree23C (Tree23C Int, Tree23C Int)`, i.e. a 2-3-tree with holes that contain changes:

```
gcp $ change x y =
  Node2C
    (Hole (Hole 0, Hole 0))
    (Node2C
      (Hole (s     , Hole 1))
      (Hole (Hole 1, Hole 1)))
```

**Closing contexts** In our example above, there is a change `(s, Hole 1)` where the hole in the insertion context does not appear in the corresponding deletion context. Such changes are called *open*. The last step of the algorithm closes any open subtrees of the spine:

```
diff x y = closure $ gcp $ change x y =
  Node2C
    (Hole (Hole 0, Hole 0))
    (Hole
      ( Node2C s        (Hole 1)
      , Node2C (Hole 1) (Hole 1)
      ))
```

This makes it easier to apply patches.

In Section 3.3, we discuss which parts of the algorithm we chose to adopt.

# 3. Specification & Design

This section gives a broad abstract overview of the concepts we implement in Chapter 4.

In Section 3.1 we introduce our algorithm for pairing bindings and explain how it can be used for Core programs and let expressions.

In Section 3.2 we explain our approach to make Core terms $\alpha$-equivalent where possible using the nominal techniques outlined in Section 2.2.3.

Finally, in Section 3.3, we show how we adopted parts of the tree-based differencing algorithm explained in Section 2.4.2 to provide structural differencing in CoreDiff.

## 3.1. Pairing

We introduce an algorithm that, given two lists of bindings, finds pairs of bindings which can be assimilated and then compared individually.

This section will build on the example given in Figure 3.1. We compiled `Fac.hs` (see Appendix A.3) with GHC 8.92de9e and GHC 9.ef9dbc and compared the set of top level bindings the first float out steps created. The exported binders, highlighted in cyan, are trivial to pair as their names only appear once in each program. The remaining binders have ambiguous names. In order to to pair them, we analyze their occurrences in bindings whose binders have already been paired.

### 3.1.1. Trivial pairings

As a baseline, we consider the exported binders of each program. These are trivial to pair up because their names are guaranteed to be distinct up until the CoreTidy phase [14]. Additionally, each remaining binder should (transitively) occur in their bound expressions as they would be useless otherwise.

Figure 3.1 shows exported binders in cyan. The set of exported binder *names* is the same for GHC 8.92de9e and GHC 9.ef9dbc (see Section 2.1). Even the order of the binders is the same. This order is subject to change across compiler versions because different versions may apply different compiler passes. Especially for bigger programs, this makes it non-trivial to pair up the remaining, non-exported binders.

### 3.1.2. Iterative pairing

In order to reliably pair up the remaining binders, we inspect occurrences in pairings that have already been found.

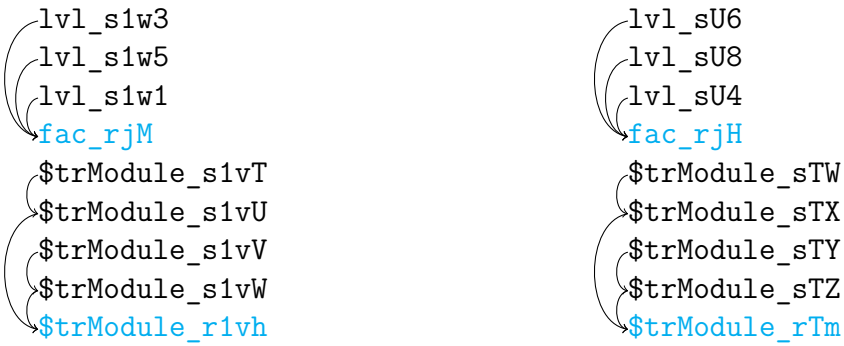


**Figure 3.1.:** Top level bindings in the output for the first float out step for `Fac.hs`, GHC 8.92de9e (left) and GHC 9.ef9dbc (right). Exported binders are highlighted in cyan. Arrows indicate an "occurs-in" relationship.

### Example for iterative pairing

Consider the bound expressions for `$trModule_r1vh` and `$trModule_rTm` which were trivial to pair because `$trModule` is an exported binder and therefore appears only once in each program. First when compiled using GHC 8.92de9e:

```
$trModule_r1vh =
  GHC.Types.Module
    $trModule_s1vU $trModule_s1vW
```

Then using GHC 9.ef9dbc:

```
$trModule_rTm =
  GHC.Types.Module
    $trModule_sTX $trModule_sTZ
```

Their bound expressions both call `GHC.Types.Module` on some variables. Because they are top level binders yet to be paired and they occur in the same place on each side in the exported binding `$trModule`, we can now pair `$trModule_s1vU` with `$trModule_sTX` and `$trModule_s1vW` with `$trModule_sTZ`. Now consider the bound expressions of the first new pair, again starting with GHC 8.92de9e:

```
$trModule_s1vU =
  GHC.Types.TrNameS $trModule_s1vT
```

Then, GHC 9.ef9dbc:

```
$trModule_sTX =
  GHC.Types.TrNameS $trModule_sTW
```

In the same way as before, we can now pair `$trModule_s1vT` with `$trModule_sTW`. In this manner, we can pair up all `$trModule`-bindings in the given programs. We found this approach to work remarkably well even for bigger programs.

### Iterative pairing algorithm

We will now generalize this procedure into an iterative algorithm which we will implement in Section 4.4.

A single iteration step is based on the function

$$\begin{aligned}&\textsc{Dis} \quad : \mathsf{Exp} \times \mathsf{Exp} \to 2^{\mathsf{Var}\times\mathsf{Var}}\\&\textsc{Dis}(e, e') = D\end{aligned}$$

which calculates the set $D$ of disagreeing occurrences of *free* binders in two expressions $e$ and $e'$. Our iteration state is a tuple $(P, U_L, U_R, F)$ where

- $P \subseteq \mathsf{Bind} \times \mathsf{Bind}$ is a set of potential pairings of bindings,
- $U_L \subseteq \mathsf{Bind}$ is the set of remaining unpaired bindings of the left side,
- $U_R \subseteq \mathsf{Bind}$ is the set of remaining unpaired bindings of the right side,
- $F \subseteq \mathsf{Bind} \times \mathsf{Bind}$ is the set of pairings whose expressions have been considered.

A single step of our iteration is performed by

$$\textsc{Step}(P, U_L, U_R, F) = (P' \cup N, U'_L, U'_R, F \cup \{(b = e, b' = e')\})$$

where

$$\begin{aligned}P &= \{(b = e, b' = e')\} \uplus P'\\D &= \textsc{Dis}(e, e')\\N &= \{(b = e, b' = e') \mid b = e \in U_L, b' = e' \in U_R, (b, b') \in D\}\\U'_L &= \{B \mid (B, \cdot) \notin N, B \in U_L\}\\U'_R &= \{B \mid (\cdot, B) \notin N, B \in U_R\}\end{aligned}$$

We extract an established, but not yet considered pairing of bindings $(b = e, b' = e')$ from $P$. Then, we find the set $D$ of disagreeing free occurrences in their bound expressions. We establish the set $N$ of new pairings to add to $P$. It contains pairings of bindings from $U_L$ and $U_R$ where the binders are in $D$. Finally, we remove the bindings that will be added to $P$ from $U_L$ and $U_R$ and add $(b = e, b' = e')$ to our output set $F$.

We also define

$$\textsc{Iter}(P, U_A, U_B, F) = \begin{cases} F & , P = \emptyset \\ \textsc{Iter}(\textsc{Step}(P, U_A, U_B, F)) & , \text{otherwise} \end{cases}$$

which applies Step until $P$ is empty. This function can be used to perform a full pairing.

#### Initial iteration state

There are two ways in which ITER is used: Pairing whole programs and pairing bindings in recursive let expressions. Depending on the use case, the initial state is chosen in different ways.

**Pairing programs** To pair the top level bindings of the programs $A$ and $B$, we define

$$\text{PAIR}_{\text{PROG}}(A, B) = \text{ITER}(T, U_A, U_B, \emptyset)$$

where $T$ are the trivial pairings described in Section 3.1.1, and $U_A$ and $U_B$ are the remaining binders in $A$ and $B$.

**Pairing recursive let expressions** To pair the bindings in two recursive let expressions, we can not use the trivial pairings approach. That is because no bindings in let expressions are exported. Instead, we define

$$\text{PAIR}_{\text{LET}}(\textbf{letrec } \overline{b_i = e_i}^{\,i} \textbf{ in } e, \textbf{letrec } \overline{b'_j = e'_j}^{\,j} \textbf{ in } e') = \text{ITER}(D, \overline{b_i = e_i}^{\,i}, \overline{b'_j = e'_j}^{\,j}, \emptyset)$$

where $D \subseteq \text{DIS}(e, e')$ are the disagreeing free occurrences of binders $b_i$ and $b'_j$ in $e$ and $e'$.

## 3.2. Binder assimilation

In Section 2.2, we described problems introduced by the naming of binders in different versions of GHC. Because different versions of GHC may give binders different uniques and sometimes even different names, textual comparisons of Core programs often falsely mark lines as different. To reduce the number of such false positives, we considered two approaches that make expressions syntactically equal where they are $\alpha$-equivalent.

### 3.2.1. De Bruijn index

In Section 2.2.2, we gave a short example of the namefree expressions introduced by de Bruijn [11]. For our use case, the first problem with this namefree representation is that it barely resembles the original terms. While names carry no operational meaning in Core, they offer readability for humans. This means that after two terms have been converted into namefree form for comparison, they need to somehow be converted back into a readable form. Compared to the nominal approach explained below, we found this reconstruction step to be too awkward to implement.

The second problem is that our comparison needs to support multiple bindings at the "same" level. For example, consider these terms:

$$e_1 = \textbf{letrec } a = b; b = a \textbf{ in } a$$
$$e_2 = \textbf{letrec } a = b; b = c; c = a \textbf{ in } a$$

A naive implementation could simply increase the depth from the rightmost to the leftmost binding. In that case, the similarilites of $e_1$ and $e_2$ are lost:

$$e_1' = \textbf{letrec } 1; 2 \textbf{ in } 2$$
$$e_2' = \textbf{letrec } 2; 1; 3 \textbf{ in } 3$$

Other approaches to mitigate this problem were prone to similar subtle bugs.

### 3.2.2. Nominal approach

The approach we favor is based on the equivalence relation $\approx$ given by Urban et al. in [12, p. 480] which we outlined in Section 2.2.3. We adopted the use of binder permutations typical for nominal algorithms.

Consider again their rule for $\alpha$-equivalence of abstraction terms [12, Fig. 2]:

$$\frac{a \neq a' \qquad \nabla \vdash t \approx (a\ a') \cdot t' \qquad \nabla \vdash a \# t'}{\nabla \vdash a.\, t \approx a'.\, t'}$$

Our preprocessing step to assimilate binders is built in a very similar way. Given two terms $a$ and $b$, we define

$$\textsc{Asim}(a, b) = c$$

which creates a third term $c$ that has the structure of $b$ but binder names and uniques from $a$ where possible. If $a$ and $b$ are $\alpha$-equivalent, $a$ and $c$ will be syntactically equivalent except for their metadata. For example,

$$\textsc{Asim}(\lambda x.\, t, \lambda x'.\, t') = \lambda x.\, \textsc{Asim}(t, (x\ x') \cdot t').$$

Given two abstraction terms, we keep the binder $x$ of the left side and exchange any occurrences of $x$ and $x'$ in the right side. Asim can be defined along the lines of $\approx$ for the trivial cases.

Asim assumes that all binders bound in $a$ do not occur freely in $b$, which is required by $\approx$. For raw Core programs, this is not necessarily correct: Uniques, even though they can be assigned differently in two programs, may overlap. If such an overlap is detected, we cannot simply rename the free binder in $b$, as that could change the program's semantics. Instead, we can rename the binder's unique in $a$. It is trivial to find a unique id that is used in neither $a$ nor $b$. If such renaming takes place, the user should be notified.

Just like with de Bruijn's namefree form, the difficulty in this approach is to find a suitable solution for recursive let expressions. In order to solve this, we pair up the bindings of the let expressions using the function $\textsc{Pair}_{\textsc{let}}$ defined in Section 3.1.2.

Similar to recursive let expressions are alternatives in case expressions. These can always be paired trivially, since each alternative has a unique constructor.

We implement Asim in Section 4.5.

## 3.3. An alternative differencing algorithm

In this section, we adopt a tree-based differencing algorithm to compare Core terms not in their textual form, but in their structure. As we remarked in Section 2.4.1, textual differencing can not distinguish relevant differences in expressions from superficial formatting artifacts. If we compare the underlying structure instead of a textual representation, we can avoid such issues. For this purpose, we offer an alternative differencing algorithm derived from the one given by Miraldo and Swierstra in [3] outlined in Section 2.4.2.

From their algorithm, we only adopt the second step: Identifying the spine. Using the full algorithm, represented by the function FullDiff, can be counterproductive in some cases, for example (changes within terms are denoted as $\#(\cdot \ / \ \cdot)$):

$$\textsc{FullDiff}(map\ f\ primes, map\ g\ primes) = \#(0 \ / \ 0)\ \#(f \ / \ g)\ \#(1 \ / \ 1)$$

The algorithm minimizes redundancy in the resulting patch by replacing common subtrees with numbered holes. For our use case, this obscures important context. It is much easier to classify the difference in these terms if we keep the surrounding terms intact:

$$\textsc{Spine}(map\ f\ primes, map\ g\ primes) = map\ \#(f \ / \ g)\ primes$$

This example shows that simply identifying the spine is already enough to generate usable diffs.

Yet we can improve their usefulness by adding a simple postprocessing step. Consider the expressions $f\ x$ and $g\ y$. Their spine is a term application; the mismatched variables on either side become changes:

$$\textsc{Spine}(f\ x, g\ y) = \#(f \ / \ g)\ \#(x \ / \ y)$$

Since all subterms of this patch are changes, we can merge the changes into a single change on the next level. We call this transformation Mend, for example:

$$\textsc{Mend}(\#(f \ / \ g)\ \#(x \ / \ y)) = \#(f\ x \ / \ g\ y)$$

While the resulting patch contains some redundant function applications, we favor this form because the result is more readable.

To wrap up, we can define our structural differencing function as

$$\textsc{Diff}(e, e') = \textsc{Mend}(\textsc{Spine}(e, e')).$$

# 4. Implementation

This chapter describes how we extracted Core programs from GHC and implemented the preprocessing transformations, pairing and differencing algorithms specified in Chapter 3 for GHC Core in CoreDiff. CoreDiff is written in Haskell, since GHC itself and the library we use to interact with it are written in Haskell too. We only give a broad overview and discuss specific challenges we encountered. However, CoreDiff is open source software; the full source code, including instructions for building it, is available at `https://github.com/pbrinkmeier/corediff` [1].

## 4.1. The big picture

The command for comparing programs using CoreDiff is

```
$ corediff diff modA.cbor modB.cbor
```

The entry point for programs is the module `Main`, which parses command-line arguments and follows these steps:

1. Import and convert the Core modules into CoreDiff data types (Section 4.2.1, Section 4.3.2).
2. Pair the bindings of the resulting programs (Section 4.4).
3. Assimilate all paired bindings (Section 4.5).
4. With the command line flag `--structural`, compare all pairs of bindings using the tree-based differencing algorithm discussed in Section 3.3. Otherwise, pretty-print all pairs of bindings and then compare them using `diff`.

## 4.2. Exporting Core from GHC

In order to apply our algorithms, we need to extract Core ASTs from GHC. An evident approach is to let GHC print out a textual version of the Core program and then parse it into an AST. But this is tedious to implement and prone to break over time as new features are added to Core. What we went with instead is a library and GHC plugin called `ghc-dump` [2].

[1] Git commit hash at the time of writing: `53335680361058201ca35899420fdd0d0ed238da`
[2] Repository: `https://github.com/bgamari/ghc-dump`

### 4.2.1. ghc-dump

`ghc-dump` is an established tool that uses GHC's plugin system [15, Compiler Plugins] to export a Core program after each Core-to-Core pass. It consists of two packages: `ghc-dump-core`, which contains a Core AST data type and the plugin itself, and `ghc-dump-util`, which contains some utility functions. The plugin is straightforward to use: The following example invocation compiles `T18231.hs` with optimizations enabled (`-O2`) and exports a Core program after each Core-to-Core pass.

```
$ ls
T18231.hs
$ ghc -O2 T18231.hs -fplugin GhcDump.Plugin
[1 of 1] Compiling T18231             ( T18231.hs, T18231.o )
Linking T18231 ...
$ ls
T18231.hi T18231.hs T18231.o T18231.pass-0000.cbor T18231.pass-0001.cbor
...
T18231.pass-0020.cbor T18231.pass-0021.cbor T18231.pass-0022.cbor
```

In addition to the usual interface `.hi` and object `.o` files, this invocation, using the plugin `GhcDump.Plugin`, also creates a `.cbor` file after each pass. These files contain Core ASTs that are serialized using the Concise Binary Object Representation (CBOR) format [16].

In order to use these files, we do not need to concern ourselves with parsing the CBOR format. The package `ghc-dump-util` provides the function `GhcDump.Util.readDump`, which we can use to directly read them into according data types.

```
import GhcDump.Ast  (Module)
import GhcDump.Util (readDump)

main = do
  mod <- readDump "path/to/mod.cbor" :: IO Module
  ...
```

A specific goal of `ghc-dump` is to provide a consistent representation of Core ASTs across a multitude of GHC versions, starting with GHC 7.10 (released in April 2015). For this reason, the module `GhcDump.Ast` from the package `ghc-dump-core` defines its own set of data types for Core ASTs. This suits our use case of comparing Core ASTs generated by different versions of GHC.

### 4.2.2. Extending ghc-dump

As we remarked above, `ghc-dump` defines its own set of data types to represent Core ASTs. A downside is that these types miss some features of the ones defined by GHC itself. Specifically, in order for us to use `ghc-dump` in a meaningful way, we needed:

- Support for GHC 8.10 and later,
- binder metadata concerning whether a binder is exported, which is important to find trivial pairings and
- binder CPR information.

Since `ghc-dump` is open source software, we were able to fork the repository [3] and implement those features ourselves.

## 4.3. Data types

The central data structure of a Core AST is the *expression*. It contains *binders* in binding sites or as variable occurrences. Those binders have *types*. Expressions also contain *bindings* (in let expressions) and *alternatives* (in case expressions).

### 4.3.1. Data types in CoreDiff

In the module `GhcDump.Ast`, these data structures are implemented by the types `Expr`, `Binder`, `Type`, `(Binder, Expr)` and `Alt`.

In CoreDiff, we chose to replicate and extend these data types. There are several reasons why we did so.

The first one is a software engineering aspect. While it leads to code duplication, providing our own data types decouples the implementation of our algorithms from the import mechanism. If, someday, we will find an alternative to `ghc-dump` or decide to write a parser for GHC output ourselves, we can simply swap out the import and conversion functions.

Secondly, common type classes such as `Eq` and `Ord` are already implemented for the types in `GhcDump.Ast`. Haskell does not let you override these instances for consistency reasons. Having to bypass these restrictions would without a doubt have led to some hacky, hard to maintain code.

The third and perhaps most important reason is that, in order to implement the structural differencing algorithm from Section 2.4.2, we need to define an according type that supports change-shaped holes for each of our data types (similar to `Tree23` and `Tree23C` in Section 2.4.2). This means we would have had to replicate the data types from `GhcDump.Ast` regardless.

### 4.3.2. An extensible AST

Our AST data types are going to be used in two roles: For simple terms without additional constructors, or as the spines of patches generated by the Spine function. In order to avoid code duplication, we employed the approach presented by Najd and Jones in [17]. They provide a way to write extensible data types by making

[3] Repository: `https://github.com/pbrinkmeier/ghc-dump`

them polymorphic over a *variant* of the data type. This approach lets us define the data structures for our Core ASTs once and use them in both roles. Common functionality such as pretty-printing can also be defined for both roles at the same time.

For example, consider our data type `XExpr` for expressions listed in Figure 4.1. This data type represents the expressions defined in Figure 2.1. An `XExpr` can be either one of two variants: The undecorated variant, `XExpr UD`, represents expression terms only. In this variant, the `XXExpr` extension constructor cannot be instantiated. That is because we chose the extension for undecorated expressions to be of type `Void`, which cannot be instantiated. For the second variant, `XExpr Diff`, which represents the spine of a structural patch for expressions, we chose the extension to be of type `Change (XExpr UD)`, i.e. a pair of expression terms. This means that this type can be an expression, or, additionally, a change of expressions.

We assign extension types to variants using the type family `XExprExtension`. A type family is a function on the type level – it maps types to other types. In this case, it maps a variant type to its extension type for expressions. Each of the other types referenced in `XExpr` has a similar type family to select its extension. This way, we can choose the extension `Change (XBinder UD)` for `XBinder Diff`, the extension `Change (XType UD)` for `XType Diff` and so forth.

Finally, we added the shorthand type alias `ForAllExtensions` for writing signatures for functions that work with extensions. For example, to write our own instance of `Eq` for an `XExpr a`, we would have to declare the following instance

```
instance ( Eq (XExprExt a), Eq (XBinderExt a), Eq (XBindingExt a)
         , Eq (XAltExt a), Eq (XTypeExt a)) => Eq (XExpr a) where
  ...
```

because comparing the subterms of expressions demands that their extensions are comparable too. With our shorthand, we can simply write

```
instance ForAllExtensions Eq a => Eq (XExpr a) where
  ...
```

which is a great improvement. This technique is discussed in detail in [17, Sec. 3.7].

### 4.3.3. Pretty-printing terms and spines

To demonstrate how these data structures can be used, we show how we implemented pretty-printing for expressions and expression spines. For pretty-printing itself, we used the package `ansi-wl-pprint` [4]. It is based on [18] by Wadler, who describes a set of combinators for functional pretty-printing.

We begin by defining a data type for pretty-printing options and a typeclass that allows us to write overloaded `pprWithOpts` functions:

[4]Hackage: `https://hackage.haskell.org/package/ansi-wl-pprint`

```haskell
{-# LANGUAGE ConstraintKinds #-}
{-# LANGUAGE DataKinds #-}
{-# LANGUAGE KindSignatures #-}
{-# LANGUAGE TypeFamilies #-}

import Data.Kind (Constraint)
import Data.Void

data Variant = UD | Diff

newtype Change a = Change (a, a)

data XExpr (a :: Variant)
  = XVar (XBinder a)
  | XVarGlobal ExternalName'
  | XLit Lit
  | XApp (XExpr a) (XExpr a)
  | XTyLam (XBinder a) (XExpr a)
  | XLam (XBinder a) (XExpr a)
  | XLet [XBinding a] (XExpr a)
  | XCase (XExpr a) (XBinder a) [XAlt a]
  | XType (XType a)
  | XCoercion
  | XXExpr (XExprExt a)

-- XBinding, XBinder, XType, XAlt

type family XExprExt a where
  XExprExt UD   = Void
  XExprExt Diff = Change (XExpr UD)

-- XBindingExt, XBinderExt, XTypeExt, XAltExt

type ForAllExtensions (constr :: * -> Constraint) a =
  ( constr (XExprExt a)
  , constr (XBindingExt a)
  , constr (XBinderExt a)
  , constr (XTypeExt a)
  , constr (XAltExt a)
  )
```

**Figure 4.1.:** Extensible Core expression data type. Excerpt from `CoreDiff.XAst`.

```haskell
import Control.Monad.Trans.Reader

data PprOpts = PprOpts { pprOptsDisplayUniques :: Bool, ... }

pprOptsDefault = PprOpts True ...

class PprWithOpts a where
  pprWithOpts :: a -> Reader PprOpts Doc
```

The `pprWithOpts` function maps terms to *documents* under some read-only context of type `PprOpts`. By using the `Reader` monad [5], we rid ourselves of the need to pass around this context explicitly every time we pretty-print a sub-term.

We can now simply write an instance of `PprWithOpts` for each of our types to implement pretty-printing for terms and extensions at the same time.

## 4.4. Pairing algorithm

In order to assimilate bound expressions and to compare programs binding by binding, we need to pair up the bindings of the given programs.

### 4.4.1. Trivial pairings

In Section 3.1.1 we remarked that exported top level bindings are trivial to pair because their names are unambiguous. In fact, there is an exception to this: If a Haskell program has an exported binding `main`, its corresponding Core program has two exported bindings with the name `main`. The user-defined one gets assigned a unique like any other binding. The second `main` will always have the special unique `01D`. `main_01D` wraps the user-defined main in a call to `GHC.TopHandler.runMainIO` [6]. This function catches global exceptions and flushes the standard streams before exiting.

Initially, this confused the trivial pairings function; after all, it assumes that exported bindings never share a name. In its current implementation `triv` in `CoreDiff.Pairing`, we explicitly pair `main_01D` bindings before we apply it.

### 4.4.2. Iterative pairing

In Section 3.1.2 we described an iterative approach that inspects disagreeing variable occurrences in already established pairings to find more of the latter. We implement this algorithm in `CoreDiff.Pairing`. To keep track of its state, we define a data type

[5] From the `transformers` package: `https://hackage.haskell.org/package/transformers-0.5.6.2/docs/Control-Monad-Trans-Reader.html#t:Reader`

[6] See `https://hackage.haskell.org/package/base-4.14.0.0/docs/GHC-TopHandler.html#v:runMainIO`

```
data PairingS = PairingS
  { toPair        :: Seq (XBinding UD, XBinding UD)
  , unpairedLeft  :: Map (XBinder UD) (XBinding UD)
  , unpairedRight :: Map (XBinder UD) (XBinding UD)
  , paired        :: [(XBinding UD, XBinding UD)]
  }
```

which stands for "pairing state" but can also be read as "pairings", which is nice. For the fields of `PairingS`, we chose data types that support specific operations we need in the implementation of Step from Section 3.1.2. The `Seq` type from `Data.Sequence` supports queue-like operations. We use these to pop the next binding from the front end and push new bindings to the back end. We use the `Map` type from `Data.Map` to index unpaired bindings by their binders, since we always refer to those by their binders. For these, it is important to know that equality of binders is implemented purely as the equality of their uniques:

```
instance ForAllExtensions Eq a => Eq (XBinder a) where
  binder == binder' = xBinderId binder == xBinderId binder'
```

For the output set we simply used a list, because this field does not need to support any operations but pushing values to one end.

Step is implemented by `step` in `CoreDiff.Pairing`:

```
step :: PairingS -> PairingS
step (PairingS pq unpairedL unpairedR paired) =
  let
    (bind@(XBinding _ e), bind'@(XBinding _ e')) :<| pq' = pq
    disagreeing = disExpr e e'
    newPairings = catMaybes $ map bothUnpaired disagreeing
    bothUnpaired (binder, binder') =
      case (unpairedL Map.!? binder, unpairedR Map.!? binder') of
        (Just b, Just b') -> Just (b, b')
        _                 -> Nothing
    (newPairingBindersL, newPairingBindersR) =
      deconPairings newPairings
```

As a first step, we pattern-match on `pq` using `:<|` to retrieve the first pairing to consider. In this pairing's expressions, we find every pair of disagreeing free binders using `disExpr`. If both binders are also unpaired in our iteration state, we add their bindings to `newPairings`. Finally, we extract the binders of these new pairings into the lists `newPairingBindersL` and `newPairingBindersR` using the helper function `deconPairings`.

```
  in PairingS
    (pq' >< Seq.fromList newPairings)
    (unpairedL `Map.withoutKeys` Set.fromList newPairingBindersL)
```

```
    (unpairedR `Map.withoutKeys` Set.fromList newPairingBindersR)
    ((bind, bind') : paired)
```

In the new state, the queue of pairings is now extended by `newPairings`. We mark their binders as paired by removing them from the unpaired maps. We add the bindings we just considered to the list of established pairings.

ITER and DIS from Section 3.1.2 are rather trivial to implement. Their signatures are:

```
iter :: PairingS -> PairingS
disExpr :: XExpr UD -> XExpr UD -> [(XBinder UD, XBinder UD)]
```

`iter` applies `step` until `toPair` is empty. `disExpr` finds a list of disagreeing free binders for two expressions.

## 4.5. Binder assimilation

To reduce superficial differences in terms, we use a nominal binder assimilation algorithm presented in Section 3.2.2 to make them syntactically equivalent where they are $\alpha$-equivalent. What makes this approach "nominal" is that we use permutation of binders instead of substitution. In our implementation, we permute only their names and unique identifiers instead of the binders themselves. This allows us to observe differences in the metadata of $\alpha$-equivalent terms.

In Section 3.2.2, we defined $\textsc{Asim}(a, b) = c$ to mean that $c$ is a term that has $b$'s structure but $a$'s binders. In order to implement ASIM, we define the overloaded function `assimilate`:

```
class Asim a where
  assimilate :: a -> a -> Reader Permutations a
```

Its signature already gives away that this function is not a direct implementation of ASIM. Recall that

$$\textsc{Asim}(\lambda a.\, t, \lambda a'.\, t') = \lambda a.\, \textsc{Asim}(t, (a\ a') \cdot t').$$

Implementing this definition directly would result in traversing $t'$ twice – once to apply $(a\ a')$ and a second time to assimilate the result and $t$. Instead of applying permutations right when we find them, we collect a list of permutations and apply them in order when needed.

In order to apply permutations to terms, we define the overloaded function `applyPerm`:

```
class Perm a where
  applyPerm :: a -> Reader Permutation a
```

along with a helper data type and functions:

```haskell
-- in CoreDiff.XAst:
data XBinderUniqueName = XBinderUniqueName T.Text Unique
  deriving (Eq)
xBinderUniqueName    :: XBinder UD -> XBinderUniqueName
xBinderSetUniqueName :: XBinderUniqueName -> XBinder UD -> XBinder UD

-- in CoreDiff.Assimilate:
data Permutations = [(XBinderUniqueName, XBinderUniqueName)]
mkPerm :: XBinder UD -> XBinder UD -> (XBinderUniqueName, XBinderUniqueName)
```

`applyPerm` uses the `Reader` monad as well, which makes it easier to use within `assimilate`. The central instance of `Perm` is the one for `XBinder UD`, which implements the permutation defined in [2, Sec. 3]. This instance applies the list of permutations to a binder from left to right and applies it to the binder's type or kind, depending on whether it binds an expression or a type. Note that in `assimilate`, we add new permutations at the end of the permutation list so that they are in fact applied last.

There is also a trivial instance `Perm (XExpr UD)` which applies the permutations to each binder and recurses into all subterms.

# 5. Evaluation

In this chapter, we will demonstrate our implementation by repeating the example from Chapter 2 using CoreDiff instead of `diff`. Thereafter we compare CoreDiff and `diff` in terms of how many differences they found in the GHC invocations of three example modules.

## 5.1. Example: `T18231.hs`

In this section, we will do the same comparison as in Chapter 2, this time using CoreDiff instead of `diff`.

For the first pass, `Desugar (after optimization)`, CoreDiff outputs:

```
m_rm7 and m_rm0 are equivalent
$trModule_rP0 and $trModule_rMB are equivalent
```

We can see that after assimilating the binders in these bindings, they are equal.

For the second pass, the initial `Simplifier` run, CoreDiff outputs:

```
$trModule_sQf and $trModule_sNX are equivalent
$trModule_sQd and $trModule_sNV are equivalent
--- a'_sRd
+++ a'_sOV
@@ -1,6 +1,8 @@
 [LclId,
  Arity=1,
- Occ=Strong Loopbrk]
+ Occ=Strong Loopbrk,
+ Str=<L,U>b,
+ Cpr=b]
 a'_sRd ::
   Int -> (Identity ((,) () Int))
 a'_sRd =
$trModule_sQg and $trModule_sNY are equivalent
$trModule_sQe and $trModule_sNW are equivalent
--- m_rm7
+++ m_rm0
@@ -1,5 +1,7 @@
 [LclIdX,
```

```
  Arity=1,
- Occ=Many]
+ Occ=Many,
+ Str=<L,U>b,
+ Cpr=b]
 m_rm7 :: State Int ()
 m_rm7 = a'_sRd
$trModule_sP0 and $trModule_sMB are equivalent
```

Here, we can see that all `$trModule` bindings have been paired correctly. In `m` and `a'`, no differences are found except for their attached metadata.

For the next pass, the first `Float out` pass, we omit `m` and the `$trModule` bindings. The part of CoreDiff's output relevant to us is the following:

```
--- a'_sRd
+++ a'_sOV
@@ -1,13 +1,12 @@
 [LclId,
  Arity=1,
- Occ=Strong Loopbrk]
+ Occ=Strong Loopbrk,
+ Str=<L,U>b,
+ Cpr=b]
 a'_sRd ::
   Int -> (Identity ((,) () Int))
 a'_sRd =
   \ s1_aQH ->
     case s1_aQH of wild_aQW {
-      I# x_aQX ->
-        a'_sRd
-          (GHC.Types.I#
-             (GHC.Prim.+# x_aQX 1#))
+      I# x_aQX -> lvl_sP3 x_aQX
     }
--- No match
+++ lvl_sP3
@@ -0,0 +1,12 @@
+[LclId,
+ Arity=1,
+ Occ=Many,
+ Str=<L,U>b,
+ Cpr=b]
+lvl_sP3 ::
+  Int# -> (Identity ((,) () Int))
+lvl_sP3 =
```

```
+  \ x_aOF ->
+    a'_sRd
+      (GHC.Types.I#
+         (GHC.Prim.+# x_aOF 1#))
```

Compared to the `diff` output from Section 2.3, it is much clearer that a new binding has been introduced: Because `lvl` could not be paired with a binding in the module generated by GHC 8.92de9e, each of its lines are marked as insertions. In `a'`, we can observe that not only its metadata changed, but also the right-hand side of the case expression's alternative. There are no interleaved lines because bindings are compared individually.

## 5.2. Comparing `diff` and CoreDiff

In order to get a broader sense of how CoreDiff and `diff` compare, we used them to find the differences in the invocations of different GHC versions. Three modules were used as examples.

### 5.2.1. Methodology

For our comparison, we compiled each example file using different compilers and different compiler options.

#### Compiler versions

We chose the same compiler versions as in the previous chapters: GHC 8.92de9e is a GHC HEAD checkout from June 2020 [1]. GHC 9.ef0dbc is the latest commit from GHC merge request !4207 from October 2020 [2]. We chose these versions because the merge request in question gives several typical examples of CoreDiff's use case.

#### Tested modules

In order to find suitable testcases, we looked through merge request !4207. We used module `T18231.hs` (listed in Appendix A.1), which we already used in previous sections, and module `T13031.hs` (listed in Appendix A.2), which also compiles differently on the given GHC versions.

Additionally, we consider another module `Fac.hs` (listed in Appendix A.3), which simply contains a faculty function. This module is used to showcase CoreDiff's behavior for compilations that behave similarly for both GHC versions.

#### Comparison methods

We use three methods to compare Core programs:

[1] Git commit hash: `92de9e25aa1a6f7aa73154868521bcf4f0dc9d1e`
[2] Git commit hash: `ef9dbc8f403194d4422a551bf6874c6d532f92bd`

**Textual difference with uniques** The first comparison invokes GHC with a selection of options that suppress some unwanted information:

```
$ ghc -O2 <module> \
      -dverbose-core2core -dsuppress-unfoldings \
      -dsuppress-ticks -dsuppress-coercions
```

We chose these options because unfoldings, ticks and coercions are not handled by CoreDiff either. The flag `-dverbose-core2core` makes GHC print the result of each Core-to-Core pass. We save the output of each pass into a single file and compare them using:

```
$ diff -u --color=always file1.txt file2.txt
```

**Textual difference without uniques** The second comparison also uses `diff`, but adds `-dsuppress-uniques` to the list of GHC options to suppress the output of uniques. We chose this comparison because it is an established approach to solve the same problems CoreDiff does.

**CoreDiff** Finally, we compare the programs using CoreDiff. To compare two passes exported by `ghc-dump` using CoreDiff, we invoke:

```
$ corediff diff file1.cbor file2.cbor
```

The files exported by `ghc-dump` are simply numbered in the order of their passes. Table 5.1 lists the passes and their file's number for invocations of GHC 8.92de9e and GHC 9.ef9dbc with the flag `-O2`.

#### Metric

In order to find the differences between these configurations, we compare the number of differences they find for each compilation pass of a given file. We do so by adding up the number of insertions and deletions a certain invocation of `diff` or CoreDiff finds.

### 5.2.2. Results

Table 5.2 shows for each Core-to-Core pass, how many differences between GHC versions each combination of test module and comparison method found. We can see that hiding uniques already reduces the number of differences found significantly.

#### T18231

Figure 5.1 shows how many differences each comparison method found for the Core-to-Core passes of `T18231.hs`. The graphs for the two textual comparisons have a very similar shape. The graph for CoreDiff differs drastically in three passes: The two

| Short | Core-to-Core pass | GHC 8.92de9e | GHC 9.ef9dbc |
|---|---|---|---|
| DS | Desugar | 0000 | 0000 |
| SI | Simplifier/initial | 0001 | 0001 |
| SP | Specialise | 0002 | 0002 |
| FO | Float out | 0003 | 0003 |
| – | Simplifier/main2 | 0004 | 0004 |
| – | Simplifier/main1 | 0005 | 0005 |
| SM | Simplifier/main0 | 0006 | 0006 |
| FI | Float in | 0007 | 0007 |
| CA | Called arity analysis | 0008 | 0008 |
| SA | Simplifier/post-call-arity | 0009 | 0009 |
| DA | Demand analysis | 0010 | 0010 |
| CP | Constructed Product Result analysis | 0011 | 0011 |
| WW | Worker Wrapper binds | 0012 | 0012 |
| SW | Simplifier/post-worker-wrapper | 0013 | 0013 |
| EX | Exitification transformation | 0014 | 0014 |
| FO | Float out | 0015 | 0015 |
| CS | Common sub-expression | 0016 | 0016 |
| FI | Float inwards | 0017 | 0017 |
| – | Simplifier/final | – | 0018 |
| LC | Liberate case | 0018 | 0019 |
| SL | Simplifier/post-liberate-case | 0019 | 0020 |
| SC | SpecConstr | 0020 | 0021 |
| – | Simplifier/post-spec-constr | – | 0022 |
| CS | Common sub-expression | 0021 | 0023 |
| – | Simplifier/final | 0022 | – |
| – | Simplifier/post-final-cse | – | 0024 |
| DA | Demand analysis | 0023 | 0025 |

**Table 5.1.:** File numbers for `-O2` Core-to-Core passes exported by `ghc-dump` for GHC 8.92de9e and GHC 9.ef9dbc. GHC 9.ef9dbc applies two more passes and applies its "Simplifier/final phase" earlier. The shorthands are used as labels for the diagrams below.

| Module | Comparison | DS | SI | SP | FO | SM | FI | CA | SA | DA | CP | WW | SW | EX | FO | CS | FI | LC | SL | SC | CS | DA |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| T18231 | `diff` | 10 | 42 | 42 | 45 | 58 | 58 | 59 | 60 | 56 | 54 | 71 | 54 | 52 | 52 | 56 | 63 | 83 | 64 | 57 | 51 | 60 |
| | `diff`, no uniques | 4 | 16 | 16 | 19 | 32 | 32 | 33 | 34 | 31 | 29 | 45 | 30 | 28 | 26 | 26 | 35 | 55 | 39 | 34 | 22 | 36 |
| | CoreDiff | 0 | 8 | 8 | 25 | 25 | 25 | 26 | 25 | 23 | 17 | 38 | 17 | 17 | 37 | 37 | 37 | 24 | 24 | 24 | 24 | 22 |
| T13031 | `diff` | 10 | 61 | 61 | 55 | 71 | 71 | 71 | 73 | 76 | 76 | 76 | 78 | 76 | 70 | 70 | 70 | 104 | 78 | 70 | 70 | 76 |
| | `diff`, no uniques | 2 | 34 | 34 | 24 | 40 | 40 | 40 | 42 | 45 | 45 | 45 | 47 | 45 | 39 | 41 | 41 | 75 | 47 | 39 | 41 | 45 |
| | CoreDiff | 0 | 28 | 28 | 76 | 72 | 72 | 72 | 71 | 71 | 71 | 71 | 72 | 71 | 71 | 71 | 71 | 71 | 71 | 71 | 71 | 71 |
| Fac | `diff` | 26 | 54 | 54 | 70 | 76 | 76 | 76 | 78 | 80 | 80 | 122 | 95 | 93 | 87 | 87 | 83 | 131 | 87 | 79 | 79 | 87 |
| | `diff`, no uniques | 2 | 6 | 6 | 0 | 6 | 6 | 6 | 8 | 8 | 8 | 6 | 8 | 6 | 0 | 0 | 0 | 56 | 8 | 0 | 0 | 8 |
| | CoreDiff | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 2 | 2 | 0 | 0 | 0 | 0 | 8 | 8 | 4 | 0 | 0 | 0 | 2 |

**Table 5.2.:** Differences found comparing the Core-to-Core passes of GHC 8.92de9e and GHC 9.ef9dbc. Each insertion or deletion count as a difference.

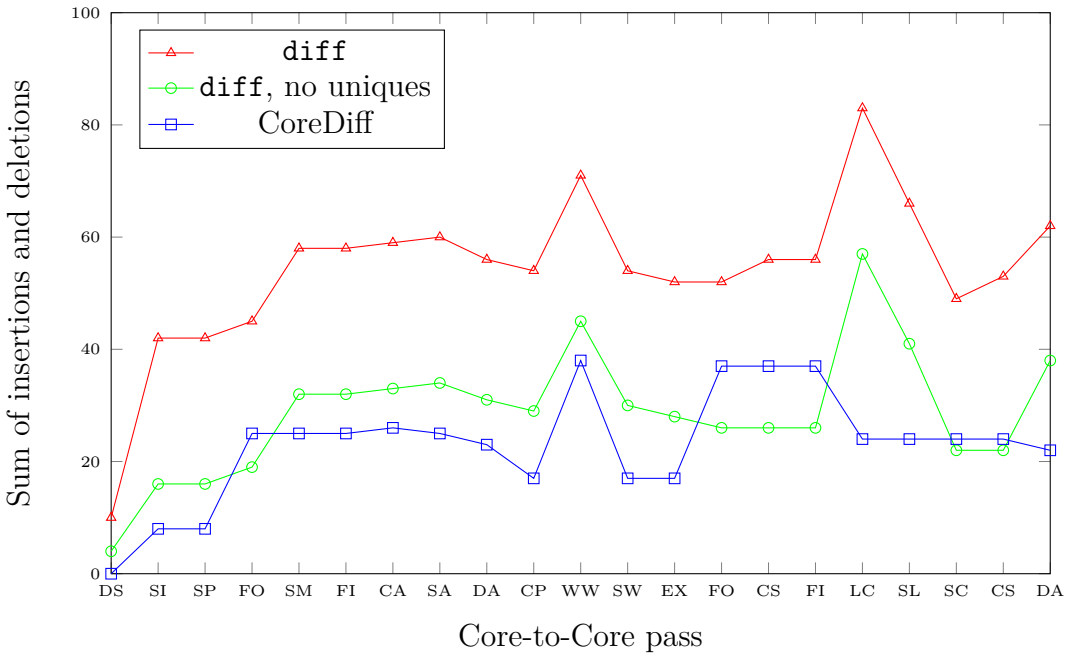


**Figure 5.1.:** Differences found in `T18231.hs`

"Float out" (FO) passes and the "Liberate case" (LC) pass. The spike of differences found by the textual comparisons is because this pass adds metadata to bindings that is not exported by `ghc-dump` and consequently not considered by CoreDiff. The reason for the contrast in the float out passes stems from the fact that CoreDiff marks each line of an unpaired binding as a difference, while the textual comparisons may interleave it with the rest of the program and find "similarities", e.g. in metadata or formatting.

For the remaining passes, CoreDiff consistently finds fewer differences than the `diff` comparison without uniques. There are two reasons for this: Firstly, CoreDiff assimilates not only uniques, but also binder names; see for example Section 2.2.1. Secondly, there is some metadata that is not exported by `ghc-dump` and therefore not considered by CoreDiff. These differences are found in the raw GHC output but not by CoreDiff.

### T13031

Figure 5.2 shows how many differences each comparison method found for the Core-to-Core passes of `T13031.hs`. We can see that for this module, CoreDiff's results are closer to the `diff` comparison with uniques. The cause is the initial simplifier (SI) step: In this step, the two versions of GHC give the binding `f` different numbers of arguments:

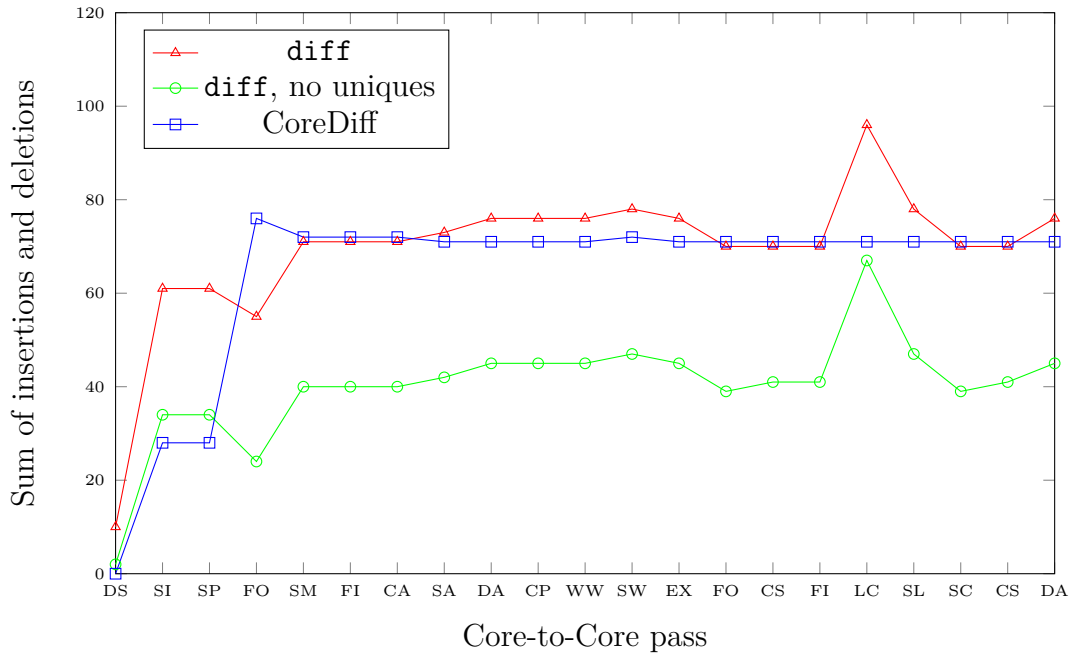


**Figure 5.2.:** Differences found in `T13031.hs`

```
 f_r3P =
-    \ ds_dy1 ->
+    \ ds_dy1 eta_B0 eta_B1 ->
```

Because of this change in `f`'s structure, CoreDiff does attempt to pair the binders within it. This is noticeable especially in the first Float out step, where terms of `f` are moved into global bindings by both versions of GHC. While `diff` without uniques matches them because they appear in the same order, CoreDiff marks all their lines as differences.

### Fac

To conclude, we present an example of a compilation that proceeds almost exactly the same in GHC 8.92de9e and GHC 9.ef0dbc. Even though this example is very simple, GHC creates three new top level bindings in the first float out step, all of which are correctly paired and deemed equivalent by CoreDiff.

As we can see in Figure 5.3, CoreDiff finds no differences for most passes from `Fac.hs`'s compilation, with two exceptions: In the first demand analysis (DA) pass, there is a change in a binder's strictness signature, which is marked as a difference by CoreDiff. In the first common sub-expression (CS) pass, both versions generate a top level binding that does not transitively occur in any exported binding. Therefore, CoreDiff marks both bindings as differences.

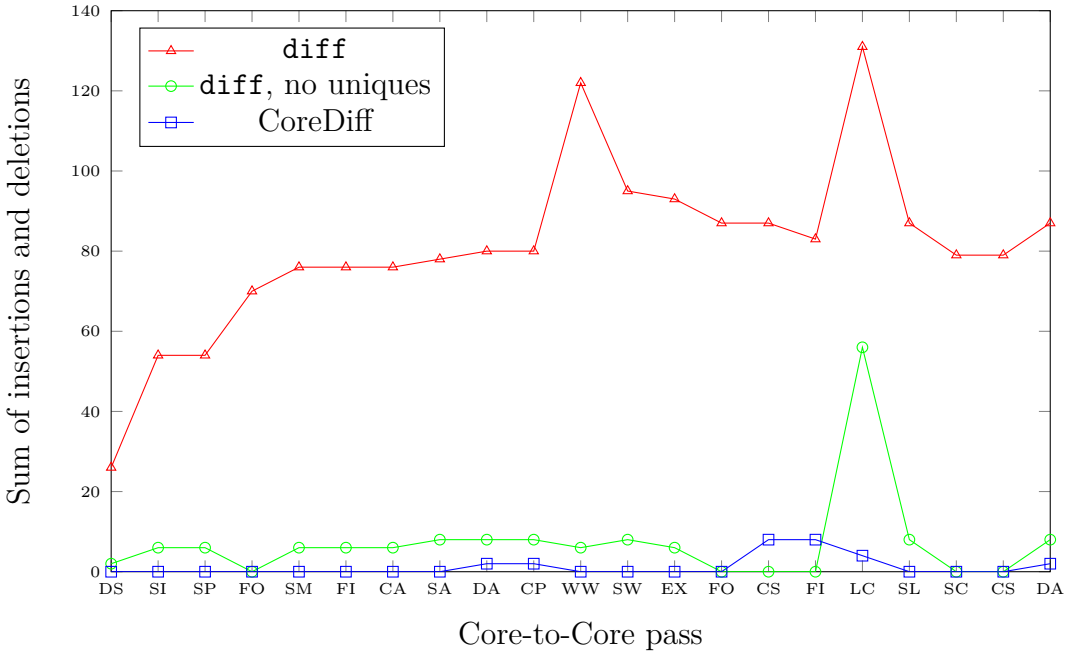


**Figure 5.3.:** Differences found in `Fac.hs`

# 6. Conclusion

In this thesis, we developed a comparison tool for GHC Core programs that improves upon purely textual differencing. In Chapter 2, we laid out an exemplary use case and the challenges it provided. In Chapter 3, we gave a broad theoretical overview of the meat of our implementation, which we described in more detail in Chapter 4. We evaluated our implementation by comparing it to the established approach in Chapter 5.

Our results for `T18231.hs` and `Fac.hs` show that CoreDiff works well for Core programs that are very similar: $\alpha$-equivalent parts of programs are reported as such while changes in metadata and structure are emphasized.

However, this emphasis on structural differences makes CoreDiff unfit for comparing Core programs that do not share the same structure. This was illustrated by example of `T13031.hs`, in which the binding `f` was compiled differently by the two tested versions of GHC.

This means that CoreDiff is suitable for the use case described in Chapter 2: Finding the first Core-to-Core pass during a compilation that produces different results for two given versions of GHC.

## 6.1. Future Work

This thesis provides a basic framework for comparing Core programs, particularly the pairing of bindings (see Section 3.1) and binder assimilation (see Section 3.2). There are multiple opportunities to improve CoreDiff's results by improving these procedures.

### 6.1.1. Fuzzy structural matching

Some terms generated by different versions of GHC are structured in similar but not quite equal ways. So far, CoreDiff considers these to be completely different from one another. For example, in `T13031.hs` we encountered a function that was given a different number of parameters by the two versions of GHC we compiled it with. This led to CoreDiff not finding pairings for top level bindings used in its body. As possible solution could be to consider curried functions as functions of multiple parameters and pairing their parameters by inspecting their bodies' disagreeing free variables. There are similar opportunities for non-exact structural matching that can be explored.

### 6.1.2. Pairing algorithm

If two versions of GHC float out terms of a program in different ways, CoreDiff is unable to pair these floated out bindings and consequently does not compare them. This could be solved by inlining non-exported bindings in such a way that the resulting terms have similar structures on both sides.

Another approach we implemented in an earlier version of CoreDiff but did not pursue much further is to offer an interactive shell where pairings can be entered manually. This could be handy for bigger programs where the current pairing algorithm is insufficient.

### 6.1.3. Tree-based differencing output

In Section 3.3, we derived a basic tree-based differencing algorithm from the one given in [3]. CoreDiff uses this algorithm to show differences in Core terms by pretty-printing their spine. This output is unfamiliar compared to the output of `diff`, which deterred us from considering it a viable alternative. Perhaps this could be solved by transforming the spine into a more familiar output format.

# Bibliography


[1] M. Sulzmann, M. M. T. Chakravarty, S. P. Jones, and K. Donnelly, "System f with type equality coercions," in *Proceedings of the 2007 ACM SIGPLAN International Workshop on Types in Languages Design and Implementation*, TLDI '07, (New York, NY, USA), p. 53–66, Association for Computing Machinery, 2007.

[2] M. J. Gabbay and A. M. Pitts, "A new approach to abstract syntax with variable binding," *Formal aspects of computing*, vol. 13, no. 3-5, pp. 341–363, 2002.

[3] V. C. Miraldo and W. Swierstra, "An efficient algorithm for type-safe structural diffing," *Proc. ACM Program. Lang.*, vol. 3, July 2019.

[4] A. Brown and G. Wilson, *The architecture of open source applications*, vol. 2. Lulu.com, 2012.

[5] S. P. Jones, K. Hammond, W. Partain, P. Wadler, and C. Hall, "The glasgow haskell compiler: a technical overview," in *Proceedings of Joint Framework for Information Technology Technical Conference, Keele*, pp. 249–257, DTI/SERC, March 1993.

[6] T. Tani, "Compiling one module: GHC.Driver.Main." `https://gitlab.haskell.org/ghc/ghc/-/wikis/commentary/compiler/hsc-main`, August 2020. Retrieved: 19 Aug. 2020.

[7] J.-Y. Girard, "The system f of variable types, fifteen years later," *Theoretical computer science*, vol. 45, pp. 159–192, 1986.

[8] R. A. Eisenberg, "System fc, as implemented in ghc," 2015.

[9] T. Tani, "Unique." `https://gitlab.haskell.org/ghc/ghc/-/wikis/commentary/compiler/unique`, June 2020. Retrieved: 30 Sep. 2020.

[10] GHC Team, "Debugging the compiler." `https://downloads.haskell.org/~ghc/8.8.3/docs/html/users_guide/debugging.html`, 2015. Retrieved: 5 Oct. 2020.

[11] N. G. De Bruijn, "Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the church-rosser theorem," in *Indagationes Mathematicae (Proceedings)*, vol. 75, pp. 381–392, North-Holland, 1972.

[12] C. Urban, A. M. Pitts, and M. J. Gabbay, "Nominal unification," *Theoretical Computer Science*, vol. 323, no. 1-3, pp. 473–497, 2004.

[13] J. W. Hunt and M. D. MacIlroy, *An algorithm for differential file comparison*. Bell Laboratories Murray Hill, 1976.

[14] T. Tani, "Data types for Haskell entities." `https://gitlab.haskell.org/ghc/ghc/-/wikis/commentary/compiler/entity-types`, June 2020. Retrieved: 1 Oct. 2020.

[15] GHC Team, "Extending and using GHC as a library." `https://downloads.haskell.org/ghc/latest/docs/html/users_guide/extending_ghc.html`, 2015. Retrieved: 12 Oct. 2020.

[16] C. Bormann and P. Hoffman, "Concise binary object reprensentation (CBOR)," RFC 7049, RFC Editor, October 2013.

[17] S. Najd and S. P. Jones, "Trees that grow.," *J. UCS*, vol. 23, no. 1, pp. 42–62, 2017.

[18] P. Wadler, "A prettier printer," *The Fun of Programming, Cornerstones of Computing*, pp. 223–243, 2003.

# Erklärung

Hiermit erkläre ich, Paul Brinkmeier, dass ich die vorliegende Bachelorarbeit selbstständig verfasst habe und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe, die wörtlich oder inhaltlich übernommenen Stellen als solche kenntlich gemacht und die Satzung des KIT zur Sicherung guter wissenschaftlicher Praxis beachtet habe.

Ort, Datum

Unterschrift

# A. Appendix

## A.1. T18231.hs

From the GHC test suite: `simplCore/should_compile/T18231.hs`.

```
module T18231 where

import Control.Monad (forever)
import Control.Monad.Trans.State.Strict

m :: State Int ()
m = forever $ modify' (+1)
```

## A.2. T13031.hs

From the GHC test suite: `stranal/should_compile/T13031.hs`.

```
{-# LANGUAGE MagicHash #-}

module Foo( f ) where
import GHC.Prim

f True  = raise# True
f False = \p q -> raise# False
```

## A.3. Fac.hs

```
module Fac where

fac 0 = 1
fac n = n * fac (n-1)
```