Karlsruher Institut für Technologie

Institut für Programmstrukturen
und Datenorganisation (IPD)

Lehrstuhl Prof. Dr.-Ing. Snelting

Test-Case Reduction for Haskell Programs

Masterarbeit von

Daniel Krüger

an der Fakultät für Informatik

```
expr2Undefined :: UT.Pass
expr2Undefined = UU.mkPass "expr2Undefined" f
  where
    f :: UT.WaysToChange (GHC.HsExpr GHC.GhcPs)
    f _ = [\_ -> GHC.HsVar GHC.NoExt
                . GHC.noLoc
                . GHC.Unqual
                $ ON.mkOccName ON.varName "undefined"]
```

Erstgutachter:	Prof. Dr.-Ing. Gregor Snelting
Zweitgutachter:	Prof. Dr. rer. nat. Bernhard Beckert
Betreuender Mitarbeiter:	M. Sc. Sebastian Graf
Abgabedatum:	16. Dezember 2020

Zusammenfassung

Mit dem Wachstum der Haskell-Programmiersprache steigt die Anzahl der gefundenen Fehler im *Glasgow Haskell Compiler*. Um letztere schnell zu beheben, benötigen die Entwickler minimale, reproduzierbare Beispiele, jedoch kostet das Reduzieren von Hand viel Zeit. Reduzierer ohne Domänenwissen können nicht alle Anwendungsfälle der Reduzierung von Haskell-Programmen abdecken. Wir zeigen, dass ein Reduzierer mit Domänenwissen bis zu 60 % kleinere Reduktionsergebnisse im Vergleich zu den Reduzierern ohne Domänenwissen erreichen kann und außerdem bis zu 70 % weniger Zeit benötigt.

The number of bugs for the *Glasgow Haskell Compiler* is constantly growing. In order to fix the latter quickly, minimal reproducible examples are required, however, manual reduction of bugs in Haskell programs takes a lot of time. Domain-independent reducers are still lacking in some respects. We show that a domain-specific reducer can produce up to 60 % smaller reduction results compared to domain-indepentent reducers while also spending 70 % less time.

Contents

1. Introduction **7**

2. Preliminaries and Related Work **11**
2.1. GHC Haskell . . . 11
2.2. Delta Debugging . . . 11
2.3. C-Reduce . . . 12
2.4. Other Test Case Reducers . . . 13
2.4.1. Hierarchical Delta Debugging . . . 13
2.4.2. structureshrink . . . 13
2.4.3. halfempty . . . 13
2.5. Perses . . . 14
2.5.1. Berkeley Delta . . . 14

3. Test Case Reducer Implementation **15**
3.1. Modular Reducer Infrastructure . . . 15
3.1.1. Search . . . 15
3.1.2. Using hsreduce as a library . . . 16
3.2. Passes . . . 16
3.2.1. Replacing Things with "Dummy" Values . . . 18
3.2.2. Removing Unused Entities . . . 19
3.2.3. Reducing To Subexpressions . . . 23
3.2.4. Exports . . . 25
3.2.5. Formatting . . . 25
3.2.6. Other Passses . . . 26
3.3. Richer Transformations . . . 26

4. Merger Implementation **31**
4.1. Preprocessing . . . 31
4.2. Existing Tools for Merging . . . 31
4.3. Requirements for Merging Haskell Modules . . . 32
4.4. Obtaining Project Information . . . 33
4.5. Accessing Renaming Information . . . 33
4.6. Essential Renaming . . . 34
4.6.1. Mapping of Names . . . 34
4.6.2. Additional Renaming . . . 35
4.6.3. Prohibited Renaming . . . 35

4.7. Applying Renamer Information 35
4.7.1. On Renamed Source 35
4.7.2. On Parsed Source 36
4.7.3. Re-Exports by “Our” Modules 36

5. Evaluation **41**
5.1. Comparison with C-Reduce 41
5.1.1. Results 41
5.1.2. Looking at Test Cases 45
5.1.3. Discussion 47
5.2. Comparison with other Related Work 48
5.3. Merging 48
5.4. Evaluation of Passes 48
5.4.1. Pass Ordering 49
5.4.2. Pass Statistics 49

6. Conclusion and Future Work **55**

A. Appendix **63**
A.1. hsreduce Implementation 63
A.2. GHC Issues 63

1. Introduction

The number of bug tickets for the Glasgow Haskell Compiler (GHC) is growing constantly. Tickets are even appearing faster than they are getting fixed, because fixing bugs takes a lot of time. One requirement to speed up debugging is for issue creators to supply minimal reproducible examples of the faulty behavior. GHC developers do not have the time to minimize bugs, they should mainly be free to develop new features and users might lack the necessary Haskell knowledge to be able to minimize their bug examples. This amount of friction might, in the worst case, incentivize users to not file bugs at all since it is too much work for everybody involved.

To illustrate this: see T18140[1] as an example. Here a developer reported a compiler performance regression when building a library. The case was minimized manually.

The key transformations were:

- merging the error causing module with the other local modules it imports
- removing all instance declarations except for the Mergeable instances
- turning the FileOptions constructor from a record into a prefix constructor
- deleting its deriving clauses
- simplyfing its type parameters to type Maybe Bool
- removing all but one language pragmas
- removing all imports

Figuring out this exact sequence of transformations requires a lot of trial and error, where after each trial the developer needs to assert that the resulting file is still a reproducer. This is very time consuming, so there would be much gained in automating this process. For this, there are already domain-independent reducers that are able to achieve up to 95 % reduction on some single Haskell files. They however lack the ability to merge Haskell projects and more advanced intermediate reduction steps like inlining functions or applying type-level functions. For the example above, they cannot perform merging of modules, transforming the a record into a prefix constructor and simplifying type parameters. Take figure 1.1 as an example, which on GHC 8.2 fails linting of the produced intermediate representation due to optimization. There, C-Reduce [1] fails to reduce the type family application, to inline a type alias, to reduce away a let-expression, to reduce to the branch of an if-expression, inline two functions and to remove the type family declaration.

Additionally, other similar tools [1] [2] [3] [4] waste a lot of time doing non-sensical reductions, for example syntactically invalid transformations. In this work, we

[1] https://gitlab.haskell.org/ghc/ghc/-/issues/18140

argue that reduction with domain-knowledge can lead to better reduction results in less time, which is why we wrote our own Haskell-specific test case reducer, called `hsreduce`, that achieves the minimal reproducer depicted in figure 1.1b.

The main contributions of this work are:

- In chapter 2 we review other test case reducers and describe in what ways they are not sufficient for reduction of Haskell programs.
- We describe in chapter 3 a modular test case reducer infrastructure that can more easily be extended by Haskell programmers than related work with additional passes for reducing Haskell programs while offering features like parallel execution of test cases and recording of pass statistics to evaluate ones efforts.
- In chapter 3 we also describe a starting set of 22 Haskell specific passes that get reduction done 30 to 80 percent faster than related work. Additionally we describe 12 richer transformations that manage to get up to 60 percent smaller reduction results compared to related work. Of the passes we describe, 31 are implemented as pure transformations of the abstract syntax tree.
- We describe merging functionality in chapter 4, to also be able to handle Haskell projects.
- In chapter 5 we evaluate our work by comparing it to C-Reduce, the best of the related work. We also look at the effect of using different pass orderings on reduction quality and examine quality criteria for passes.

```haskell
{-# LANGUAGE RankNTypes #-}
{-# LANGUAGE TypeFamilies #-}
module Bug where

j _ = h

type family G a where
   G () = Int

data Id a
instance Functor Id
type AnotherType = Bool
newtype Y f a =
  MkY (forall b.
    AnotherType -> f b)
instance Functor (Y f)

hm :: Id (G ())
hm = (j undefined) ()

weird :: Y f a ->  f b
weird (MkY g) = g True

x :: Functor g => g (G ())
x =
  let y = weird x
  in case True of
       False ->
         weird
           (weird $ weird y)
       True ->
         if 1 < 2
           then
             weird $
               (weird $
                 (weird
                   (weird x)))
           else
             weird $
               (weird $
                 (weird y))

h _ = x
```

(a) original file

```haskell
{-# LANGUAGE RankNTypes #-}

module Bug (hm) where

data Id a
instance Functor Id
newtype Y f a =
  MkY (forall b. () -> f b)
instance Functor (Y f)

hm :: Id (Int)
hm = x

weird (MkY g) = g undefined

x :: Functor g => g (Int)
x = weird $ weird $ weird (weird x)
```

(b) hsreduce's result

```haskell
{-# LANGUAGE RankNTypes #-}
{-# LANGUAGE TypeFamilies #-}
module Bug where

c d = e

type family G a
data Id a
instance Functor Id
type AnotherType = Bool
newtype Y f a =
  MkY (forall b.  AnotherType -> f b)
instance Functor (Y f)

h :: Id (G ())
h = c undefined 0

i (MkY g) = g True

j :: Functor g => g (G ())
j =
  let k = j
  in if 1 < 2
    then i (i (i j)))
    else   k

e d = j
```

(c) C-Reduce's result

Figure 1.1.: C-Reduce's limits

2. Preliminaries and Related Work

2.1. GHC Haskell

The Haskell AST (abstract syntax tree) is really vast, it consists of a large number of types of elements, like one can see from the example code 2.1 showing a small, simplified selection. Hand-written traversal would take too much time to implement and would be very error-prone, because the types are very nested and if the API changes, those hand-written traversals might break easily. That is why datatype generic traversal (more general version of the visitor pattern) was used, where we just describe the change to be made, without having to write the traversal. Since the Haskell AST implements the `Data` type class (described in “Scrap Your Boilerplate” [5]), it allows us to apply a transformations to all elements of a certain type. We make use of this generic traversal when writing our passes. It allows us to describe our changes for elements of a certain type and to call those generic traversal functions later to apply those changes.

2.2. Delta Debugging

There has already been plenty of work into the field of simplifying erroneous programs. All of the related work are instances of delta debugging where, given a interestingness test shell script and a test case. The interestingness test shell script outputs an exit code telling whether the current state of the test case exhibits some interesting behavior, for example a compiler crash. They all perform simplifications on the test case and constantly check to see, if the test case still exhibits interesting behavior despite the simplification.

`ddmin` [2] by Zeller et al. was the work that created the field of delta debugging. They see a test case as a set of changes applied to a working, normal file, which leads to observable faulty behavior. Zeller et al. describe a structure agnostic reducer that performs a kind of binary search on the input, finding the smallest relevant part. It starts with granularity two, that means it splits the test case into two halfs and checks, if any of them is interesting. If a half is interesting, ddmin tries to reduce to that subset. If it is not, ddmin tries to reduce to the complement. If that is also not successul, ddmin tries to double the granularity. If granularity `n` is already reached, it means ddmin already tried reducing to each individual change or their complement, so there is nothing else to try. They describe a test case to be 1-minimal, if deleting any character in it leads to the test case not exhibiting that faulty behavior one is after. They also show that their described ddmin algorithm

```
data HsModule = HsModule ModuleName [Export] [Import] [Decl]

data Decl = ValD Bind | TyClD ... | SigD ... | SpliceD ... | ...

data Bind = FunBind Name Matchgroup Expr | PatBind ... | ...

data Matchgroup = MG [Match]

data Match = Match MatchContext [Pat] GRHSs

data GRHSs = GRHSs [GRHS] LocalBinds

data GRHS = GRHS [GuardStmt] Expr

data Expr = Var ... | Lit ... | Lam ... | LamCase ... | App ... | ...

-- <approximately 121 more>
```

Figure 2.1.: An excerpt of the Haskell abstract syntax tree (simplified data types for better readability).

really leads to 1-minimal test cases. We did not consider ddmin further, because ddmin only does character-based deletions, while other tools also do line-based and token-based deletions.

2.3. C-Reduce

`C-Reduce` by Regehr et al. [1] is mainly a reducer for C and C++ programs. It is a modular reducer with a pluggable set of passes, which are extendable. It tries to apply all transformations until a fixpoint regarding the size in bytes is reached. In `C-Reduce`, a transformation is an iterator that walks through the test case performing changes. A transformation implements three functions: *new* which returns a new state object, *transform*, which takes a state and path to the test case, it then modifies the test case returns the resulting status code and third *advance*, which takes a state object and a path to the test case and advances to the next location.

`C-Reduce` consists of five kinds of transformations: The first are ones that operate on a contiguous segment of the tokens within a test case (changing identifiers and integer constants to 0 and 1, etc.). The second are those that make localized but non-contiguous changes (removing balanced parentheses anc curly braces, etc.). The third closely follows Berkely delta: it removes one or more contiguous lines from a test case. The number of lines to remove is initially the number of lines in the test case, and is successively halved until it reaches one line, at which point the test case is reformatted using topformflat. Fourth, it invokes external pretty-printing

commands.

Lastly, `C-Reduce` consists of a large number of C-specific passes like removing a level of indirection from a pointer- or array-typed variable, factoring a function call out of a expression or removing an unused function.

`C-Reduce` also works really well for other languages using the `--not-c` flag, where it does line and token based deletions and none of the richer, domain specific reductions for C code.

2.4. Other Test Case Reducers

2.4.1. Hierarchical Delta Debugging

Mishergi et al. [6] proposed Hierarchical Delta Debugging (`HDD`) to also make use of the fact that program code exhibits a tree structure, which can be exploited to make more structure focussed reductions. `HDD` goes in a top-down fashion through the input tree, tries to remove nodes at the currently visited level and tags the nodes of the next level for future visitation. It is also a generic reducer, they showed that it can be used for C programs, XML and video codes. But it cannot be used for Haskell, since the only implementation of it that we could find [7] relies on it being supplied a grammar, but there is currently no formal grammar for Haskell2010 with GHC extensions.

2.4.2. structureshrink

`structureshrink` by MacIver et al. [3] is a generic reducer. At its core, structureshrink uses the ddmin algorithm [2] but it applies it in a different way. It extracts a list of ngrams, which either appear at least a certain number of times in the test case or have been previously useful. Then it does two passes for each ngram: In the first pass, it splits the test case by occurrences of the ngram. So for example, for an ngram with two occurrences we would split the file into two halves. Then it tries to minimize the sequence of splits. In the second pass, it again splits the test case by occurences of the ngram. Then it tries to shrink the ngram bytewise such that joining the splits still leads to an interesting test case. Lastly, it tries to sequence minimize the whole file bytewise. It is slower by approximately a factor of five to ten than C-Reduce. Also, when running it on some Haskell test cases, it produced non-reproducing minimizations. This is why it wasn't considered further.

2.4.3. halfempty

`halfempty` [4] is another domain independent reducer, which is focused on parallelization of reduction. They solve it by building a binary tree of the possible bisection steps and then testing those different steps, assuming that most will fail to result in interesting reductions. We tried running it with the command

`halfempty ./interesting.sh Bug.hs`. On at least two test cases it produced a very small file with a file size near zero bytes where the example wasn't interesting anymore.

2.5. Perses

`Perses` by Sun et al. [8] is kind of a successor to `HDD`. It additionally takes the programming language's context free grammar as input, transforms it into the "Perses Normal Form" (a restricted form of the extended Backus-Naur form) and does transformations on a test case, based on that normal form. This way it can ensure to only do syntactically valid reductions and also supports richer transformations than HDD. In the Perses normal form, all rules have the following form:

1. $A ::= B_1^*$
2. $A ::= B_1^+$
3. $A ::= B_1^?$
4. $A ::= B_1 B_2 \dots B_n$
5. $S ::= \epsilon$

For Kleene-Star, Kleene-Plus and Optional nodes (rules of form one, two and three), Perses uses ddmin to delete their children. For regular nodes (rules of form four), Perses tries to replace them with one of their children. Perses terminates, when no tree element can be removed anymore. Perses cannot be applied because so far as they do not support Haskell and that will probably not change soon since there is currently no formal grammar for Haskell2010 with GHC extensions. Even if Perses supports Haskell one day, it cannot do richer transformation like inlining functions or normalizing type family applications.

2.5.1. Berkeley Delta

Regehr et al. also mention [1] a generic line based delta debugging approach by McPeak and Wilkerson, but we were unable to find out more about it. This project might be not maintained anymore. It is supposed to produce variants by removing one or more lines from the input.

3. Test Case Reducer Implementation

After looking at related works, we saw that they either cannot be applied at all or do not have domain-specific passes for Haskell. So we decided to develop our own tool, called hsreduce, for reducing Haskell programs[1] , to examine the question: do domain-specific passes bring better reduction? In this chapter we go over our modular test case reducer infrastructure, the starting set of Haskell passes and the richer transformation that we wrote. We will write often write reducer in short for test case reducer.

3.1. Modular Reducer Infrastructure

hsreduce is a modular reducer with a driver module and several pass modules.

As an overview, hsreduce consists of:

- the main module, which plugs the command line arguments into the driver module
- a driver module
- several pass modules
- an util module with helper functions that are used by almost all passes
- a parser module for getting the parsed, renamed and typechecked source
- a merging module, to be able to merge Haskell projects

The driver module is the heart of hsreduce. It expects as input a list of passes, as well as the paths to the interestingness test shell script and the test case. Next, it does some initilization, mainly parsing the test case and building an initial state. The driver module then calls all passes each round and terminates, when no pass was able to be applied in the current round or the changes did not result in a new and different abstract syntax tree (AST). Every round the passes see the current AST and return their calculated proposed changes. Those changes are then divided onto the thread pool, tested if they result in interesting changes and then applied.

3.1.1. Search

Our approach is greedy, the first transformation possible is applied. This might lead to the possibility to only reach a local maximum regarding the end file size, because

[1] https://github.com/dnlkrgr/hsreduce

a better transformation might be disabled by earlier transformations. On the other hand: true breadth-first search might be too expensive because all changes should be looked at in parallel and should be followed; this seems to explode in the space requirements because all the temporary results of those paths would needed to be saved in parallel.

3.1.2. Using hsreduce as a library

It is very easy to extend hsreduce. As an example, let us say we want to add a pass that tries to turn every expression into the `undefined` expression. In figure 3.1 it is shown, how we could do that. In that example, `myHsreduce` has all the functionality of hsreduce plus this added pass for handling expressions. Now, myHsreduce would check for any occurring expression in an input Haskel program, whether it can be turned into `undefined`. Also, when adding a pass, statistics for that pass are also automatically recorded. We think this speaks for a certain quality of our approach. Users do not have to think about parsing, how to apply passes on the AST and other boilerplate tasks. They can concentrate on thinking about how certain elements can be simplified and write it down in a straightforward fasion, almost like a reduction rule. We think because it is that easy to add passes that it speaks for a certain quality of our architecture. Since almost all users of a Haskell reduction tool are Haskell programmers, having a tool that is written in Haskell might make it easier for them to contribute, compared to writing passes in Perl like they would have to if they wanted to expand C-Reduce.

3.2. Passes

The reduction of hsreduce consists of three major types of functions: local changes, passes and actions.

Local changes are at the lowest level. With them we specify for a certain kind of Haskell AST element, in what ways it can be reduced. In our example figure 3.1, that is the local function `f`, which shrinks values of type GHC.HsExpr GHC.GhcPs (i.e. parsed Haskell expressions). The type of local changes is a -> [a -> a].

But local changes do not suffice. To get a new program we have to apply the change to the AST. This AST changing function we call passes. They encapsulate a local change and lift it into the context of ASTs, a function of type AST -> [AST -> AST]. This encapsulation is done by the `UU.mkPass` function in figure 3.1. First, they get all the values of that exact type the local change is operating on. Second, they apply the local change on these values, this gives a list of functions. The functions in this list are then composed with a helper function to lift those functions to become global AST transformations (which only do one change at a precise location).

Passes that work on the renamed AST or the typechecked AST can also be added, but were not needed so far. In contrast to earlier work, here the passes are only

```haskell
module ExtensionExample where

-- imports from the ghc library
import GHC
import OccName as ON
-- imports from hsreduce
import Reduce.Driver as RD
import Reduce.Passes as RP
import Util.Util as UU
import Util.Types as UT

myHsReduce shellScriptPath testCasePath =
    RD.hsreduce
        [mapM_ UU.runPass (expr2Undefined : RP.allPurePasses)]
        1                -- number of threads
        shellScriptPath
        testCasePath
        True             -- record statistics

expr2Undefined :: UT.Pass
expr2Undefined = UU.mkPass "expr2Undefined" f
  where
    f :: UT.WaysToChange (GHC.HsExpr GHC.GhcPs)
    f _ = [\_ -> GHC.HsVar GHC.NoExt
                 . GHC.noLoc
                 . GHC.Unqual
                 $ ON.mkOccName ON.varName "undefined"]
```

Figure 3.1.: Example extension of hsreduce.

responsible for producing variants; nothing else. Also to our knowledge, this is the first time that reduction passes are implemented as pure functions.

Passes alone also do not suffice. We additionally have to get the old state, apply our change on the AST of the current state, then write that change to a file and run the interestingness test shell script. For that, we have actions in our custom monad transformer, in which we store our configuration and state. With actions, we take passes and encapsulate them into our monadic context. In figure 3.1 this is done by the expression `UU.runPass`. First, we get the old state and its parsed source. Then we apply the pass on it, getting a list of proposed changes. For each proposed change we see if it is interesting by writing the result to a temporary file and running the interestingness test shell script on it; if the result is interesting, we update our state.

As said earlier, Haskell specific passes are one of the main contributions of this work. Now follow the individual passes in more detail. We go over a majority of features from the Haskell language and explain the passes that were implemented to handle them.

3.2.1. Replacing Things with "Dummy" Values

This is our first pass: it replaces any expression by simply `undefined`, which is suitable because it is of polymorphic type, that means it can be plugged in anywhere and when evaluated throws an error. This is a good way for us to test, whether an expression is needed or not. If it is needed, it will be evaluated and having turned that expression into `undefined` means that the program will just crash. We use the generic traversal capability of the GHC AST to make it a function that operates on Haskell expressions. That means for any expression, we check if we can turn it into `undefined`. figure 3.2 shows examples of expressions where this transformation can happen. Adding this pass gives us the ability to test for every expression if it is needed. If not, we can turn it into this “dummy” value and free up the references in this expression. This then enables later passes like removing matches or removing declarations.

Next, we try to turn any type into the unit type. If it is corresponding value is undefined or unevaluated this succeeds and can delete references on declared types. We also try to turn types into the wildcard type. They both have almost the same effect, i.e. freeing up types. We think turning types into unit is a better pass because turning types into the type wildcard might make specialized types polymorphic, which might decrease reduction performance. Also, turning types into the type wildcards might not be usable so often because it requires the PartialSignatures extension.

Lastly, hsreduce tries to turn patterns into the wildcard pattern. This might free up references on constructors, which in turn might enable further type simplifications.

Pass 1: Turn Expression into undefined For each expression, try to turn it into undefined.

Pass 2: Turn Type into Unit For each type, try to turn it into the unit type.

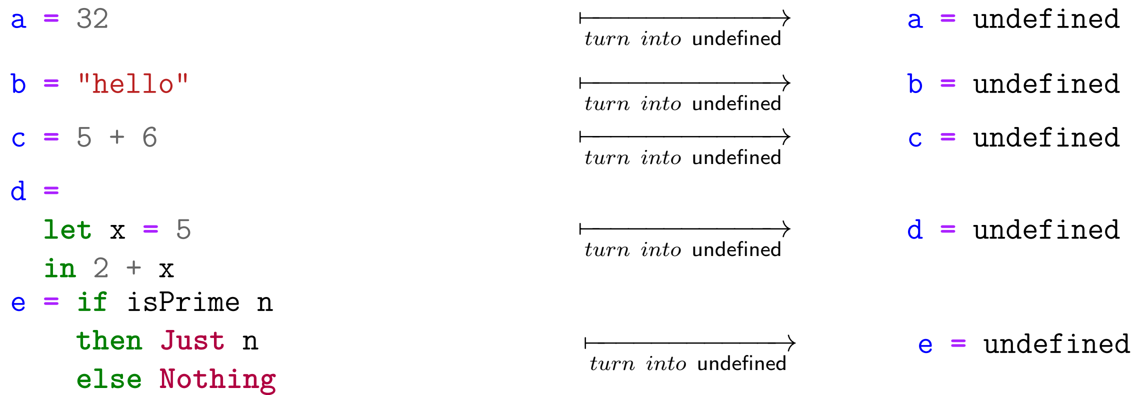

Figure 3.2.: Turning expressions into `undefined`.

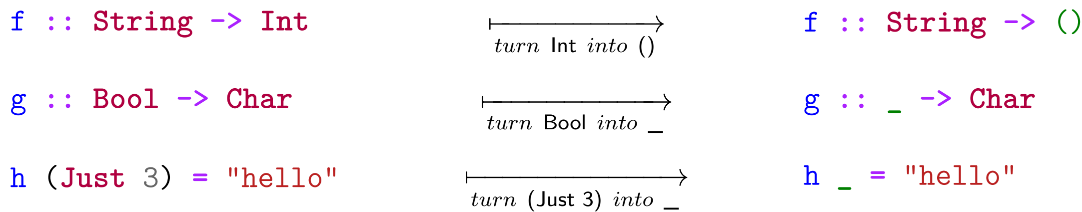

Figure 3.3.: Turning things into dummy values and handling declarations.

Pass 3: Turn Type into Type Wildcard For each type, try to turn it into a type wildcard.

Pass 4: Turn Pattern into Wildcard Pattern For each pattern, try to turn it into the wildcard pattern.

3.2.2. Removing Unused Entities

Declarations

Now that we have a way to turn values into dummy values, which frees a lot of references, we now also need a way to delete declarations. Before we can delete declarations though, we need to also delete their type signatures. Deleting a declaration which still has a type signature is a useless transformation, since it will always fail because we cannot have stand-alone type signatures. Here one has two options: to delete type signatures while deleting declarations or removing type signatures separately. We opted to do it separately because even if we do not delete the declaration, removing the type signature might still free references on types. But then we face another problem: multiple identifiers can share the same type signature, which can then make it impossible to remove the signature because it might be necessary for other identifiers.

So first, we split type signatures. This operation should always work, and it is a semantic preserving transformation.

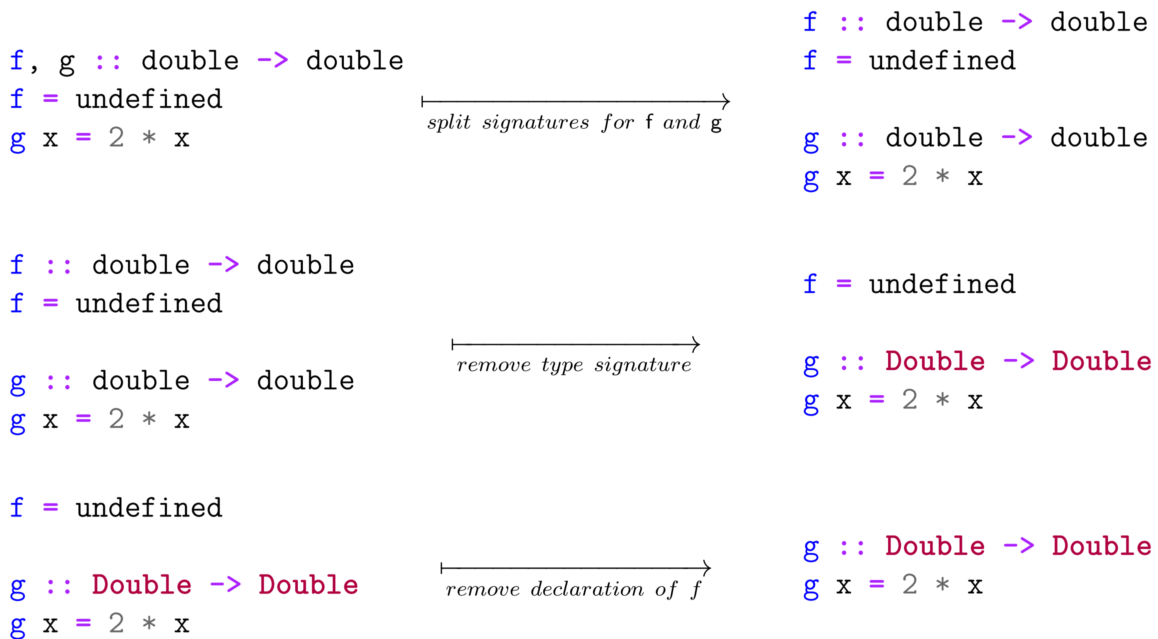

Figure 3.4.: Splitting signatures, removing signatures and removing declarations.

Next, we try to remove signatures. The signature type also includes fixity declarations and several types of pragmas, like inline and specialize pragmas. If we try to remove a declaration first we might run into the problem that the signature is still around and without a corresponding declaration. Removing signatures should be possible most of the time because the types can be inferred anyway. This might not be a semantic preserving transformation, because it might give expressions more polymorphic types than they had.

Lastly, we try to remove unused declarations, one by one. This can be any declaration, whether it be a data declaration, type synonym, class, instance and else. This is of course not a semantic preserving transformation. For unused declarations we call GHC with flags to show warnings for unused bindings. If nothing shows up, we try brute-force to delete all possible declarations.

For all three passes, see figure 3.4 for an example.

Pass 5: Split Type Signatures If a type signature cotains multiple identifiers, try to split it into multiple type signatures.

Pass 6: Remove Type Signatures For each type signature, try to remove it.

Pass 7: Remove Declarations For each declaration, try to remove it.

Data Declarations and Data Types

After running out of declarations to remove, we have to step further down. One of the the most common declarations are data declarations. For data declarations, all

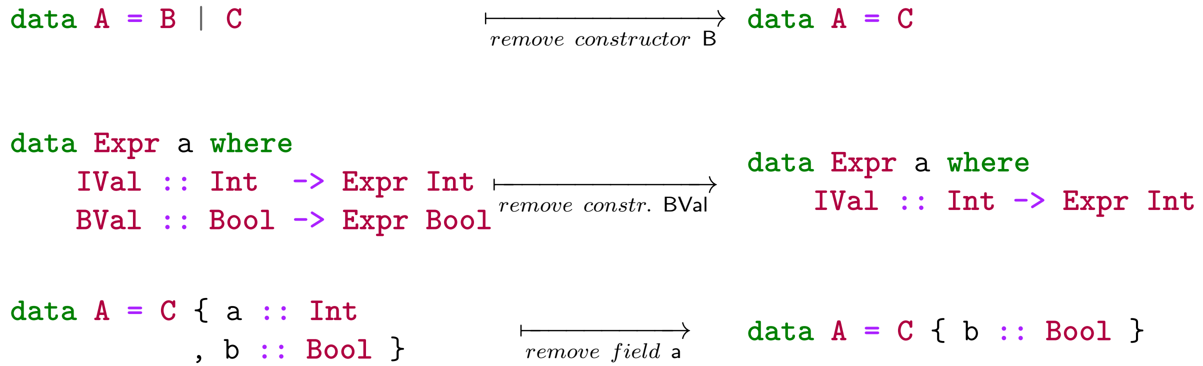

Figure 3.5.: Removing unused constructors and unused fields.

unused constructors are filtered out. This is of course a destructive pass. Additionally, we see if we can simplify the constructors in declarations. For record constructors we try to remove unused fields. For generalized algebraic datatypes (GADTs), we try to remove qualifying variables. For both passes, see figure 3.5 for an example.

Pass 8: Remove Constructors For a data declaration, try to remove each of its constructors.

Pass 9: Simplify Constructors For a constructor, if its a record constructor try to remove each of its fields. If its a GADT constructor, try to remove qualifying variables.

We also thought of turning GADTs into algebraic datatypes (ADTs) but thought this wouldn't make sense most of the time, because GADTs are used exactly to have type distinctions between the several constructors.

Functions

After turning expressions into `undefined` there are often some reduction opportunities in functions. For functions, we have three things where reductions can be done: If there are unused matches in a function then we can remove them. This most ofen happens if the right-hand side of the match is equal to the `undefined` expression. After matches, the next big thing to reduce are right-hand sides (RHS), which a match can have multiple of. Furthermore, multiple of the latter might be unused. Some guards might be garbage and can be collected.

Pass 10: Remove Matches For a function, try to remove each of its matches.

Pass 11: Remove Right-Hand Sides For a match, try to remove each of its right-hand sides.

Pass 12: Remove Guards For a right-hand side, try to remove each of its guards.

For all these three one can see examples in figure 3.6.

```
f _          = undefined                          f (Left s) = s
f (Left s) = s             ⟼ (remove match)
```

```
f
    | undefined = expr1    ⟼ (remove RHS)         f | otherwise = expr2
    | otherwise = expr2
```

```
f x
    | 1 < 2                ⟼ (remove guard)       f x | x == Just 3 = True
    , x == Just 3 = True
```

Figure 3.6.: Removing matches, right-hand sides and guards.

Expressions with Lists of Subexpressions

We also handle all expressions that have lists of subexpressions. We try to filter out all unnecessary subexpressions. Examples of expressions with lists of subexpressions include record update expressions, explicit tuples, list expressions, case expressions and multi-way-if expressions. See figure 3.7 for an example.

For example when removing field updates, this might free up those fields for removal from their constructor. For case expressions and multi-way if-expressions, we also try for each of their branches to replace them by their branch.

Pass 13: Filter out List of Subexpressions
Take an expression with a list of subexpressions and try to filter them out, one by one.

Miscellaneous

First, we try to remove functional dependencies. This might tell us, if the behavior stems from functional dependencies.

Then, we try to remove result kind signatures from type families. If the kind annotation is just *Type* or ***, it is redundant information.

For deriving clauses, we delete type classes from them or try to remove the whole clause.

For all three one can see examples in figure 3.8.

Pass 14: Remove Functional Dependencies For each functional dependency, try to remove it.

Pass 15: Remove Deriving Clause For each deriving clause, try to remove it.

Pass 16: Filter out Typeclasses from Deriving Clauses
For a deriving clause, try to remove each of its typeclasses.

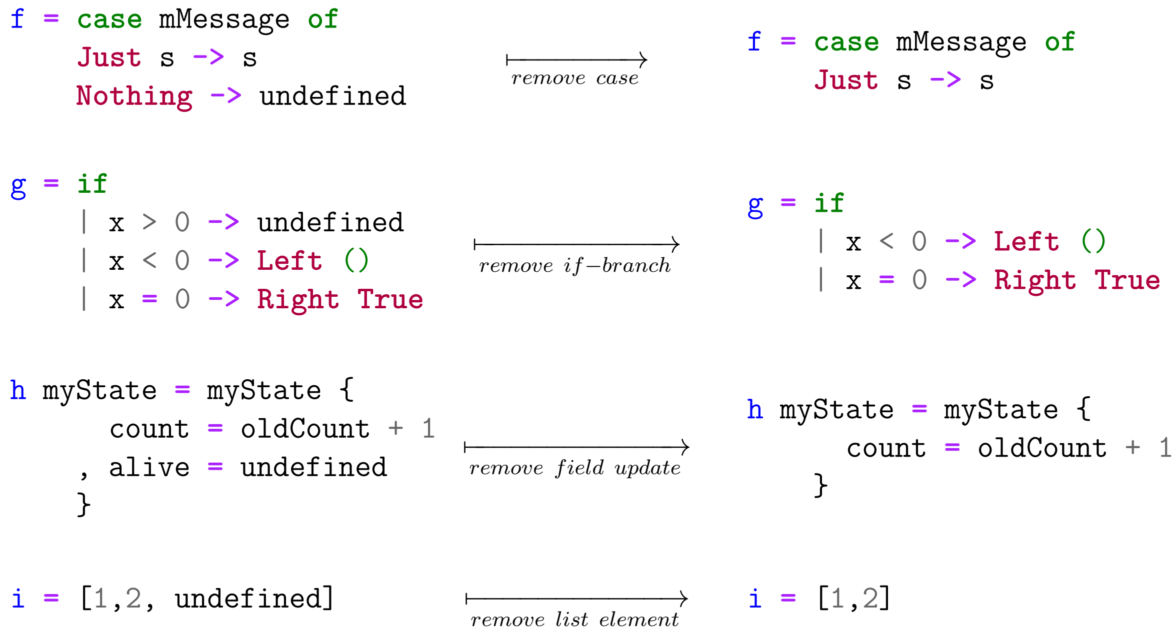

Figure 3.7.: Simplifying expressions with lists of subexpressions.

Pass 17: Remove Result Kind Signature For each type family, try to remove its result kind signature.

3.2.3. Reducing To Subexpressions

Expressions

Additionally, we look at simplifying expressions, which most often means reducing to one subexpression. We try to turn if-expressions into one of their branches, try substituting case-expressions with one of their cases, turning application of two expressions to one of the two and turning an operation involving two expressions into one of the two. Moreover, we try to remove type annotations, type applications and strictness annotations.

For example in an if-expression we try to substitute it to one of its branches. If that succeeds, it frees up the expression in the head of the if-statement.

Other types of expressions that are reduced: arithmetic expressions, list and tuple expressions.

For examples, see figure 3.9.

Pass 18: Simplify Expression For each expression, try to replace it by one of its subexpressions.

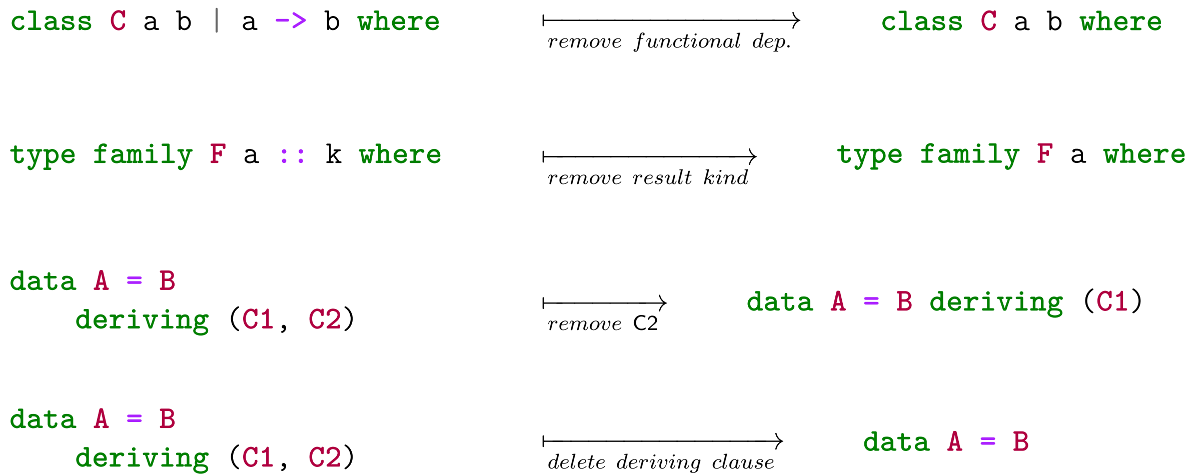

Figure 3.8.: Removing functional dependencies, kind annotations and handling deriving clauses.

Figure 3.9.: Reducing to subexpressions.

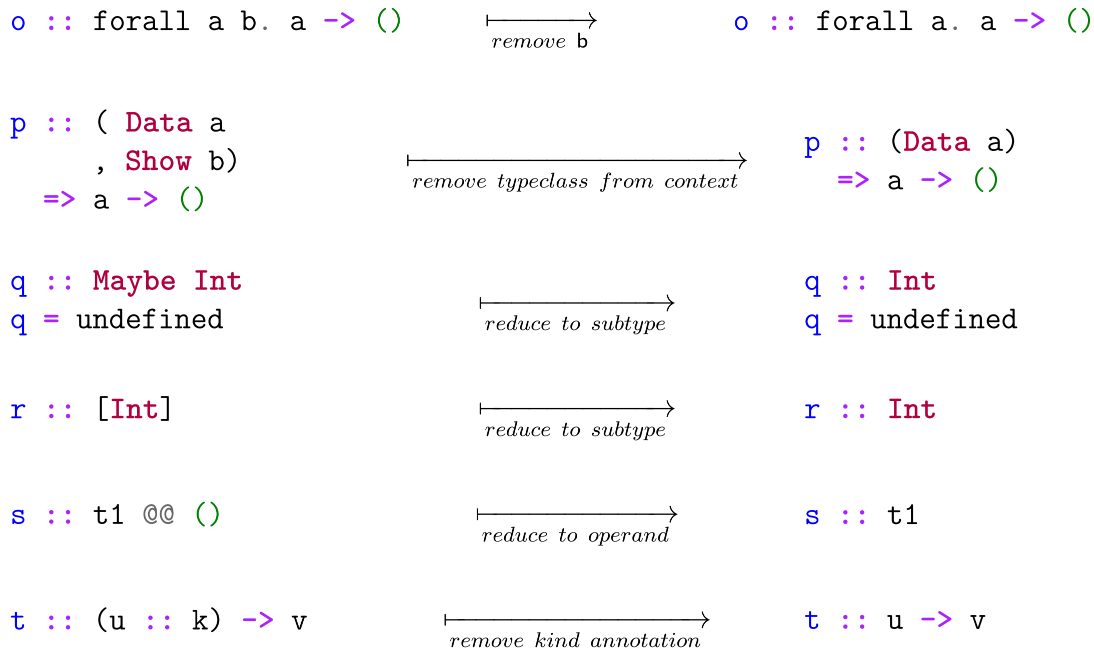

Figure 3.10.: Reducing to subtypes.

Types

Next, hrseduce tries to simplify types. We try to reduce contexts, qualified type variables in foralls, removing kind signatures and turning applications and type operations into one of the subtypes. See figure 3.10 for examples.

For example with a type operator we try to reduce to one of its operands. This might free up the other operand.

Pass 19: Simplify Type For each type, try to replace it by one of its subtypes.

3.2.4. Exports

If exports are explicitly listed, hsreduce tries to delete unused exports.

If no exports are specified, everything is exported implicitly. We turn the implicit export-alls into explicit export-alls. hsreduce takes all declaration names and adds them to the list of exports. Type operators need special treatment, there we need to add the keyword "type" to disambiguate type and normal value operators.

Pass 20: Remove Exports If exports are implicit, make them explicit. Then for each export, try to remove it.

3.2.5. Formatting

Using the GHC-API, we get the AST representation of the test-case. When printing that represention we also get rid of confusing formatting which the input might have

had. Unnecessary white space and comments are thrown away in the process.

3.2.6. Other Passses

Additionally, hsreduce includes these passes, which are too small to be worthy of their own section.

Pass 21: Remove Imports For each import, try to remove it.

Pass 22: Remove Pragmas For each pragma, try to remove it.

3.3. Richer Transformations

The previous passes could mostly be written as a local change lifted into the context of transforming ASTs. The following transformations necessitate several elements of the AST to be changed at once.

Removing Unused Parameters

Removing parameters / arguments can be done for different types of Haskell language features:

Pass 23: Remove Function Parameters For a function, its type signature and its usage sites: try to remove each of its parameters.

Pass 24: Remove Method Parameters For a method, try to remove each parameter from both the typeclass and its instances.

Pass 25: Remove Constructor Arguments For a constructor, try to remove arguments from it and its usage sites.

Pass 26: Remove Type Family Parameters For a type family, try to remove parameters from both the declaration and usage sites.

Pass 27: Remove Typeclass Parameters For a multi-parameter typeclass, try to remove each of its parameters.

They all share a common reduction helper function. First, we need a way to get a list of all names of elements that have parameters and their respective parameter counts. Then, we map over a list of the indices of the parameters for each name and see if we can remove it. But before we can do that, we need to check the number of indices again, since it might have decreased because we were able to delete earlier indices. Finally, we can run a custom transformation on the AST. The transformation needs to change several parts in the AST at once. For example for functions, we need to remove a parameter from the binding, type signatures and expressions. Otherwise we would receive an error that there is a mismatch in the number of parameters.

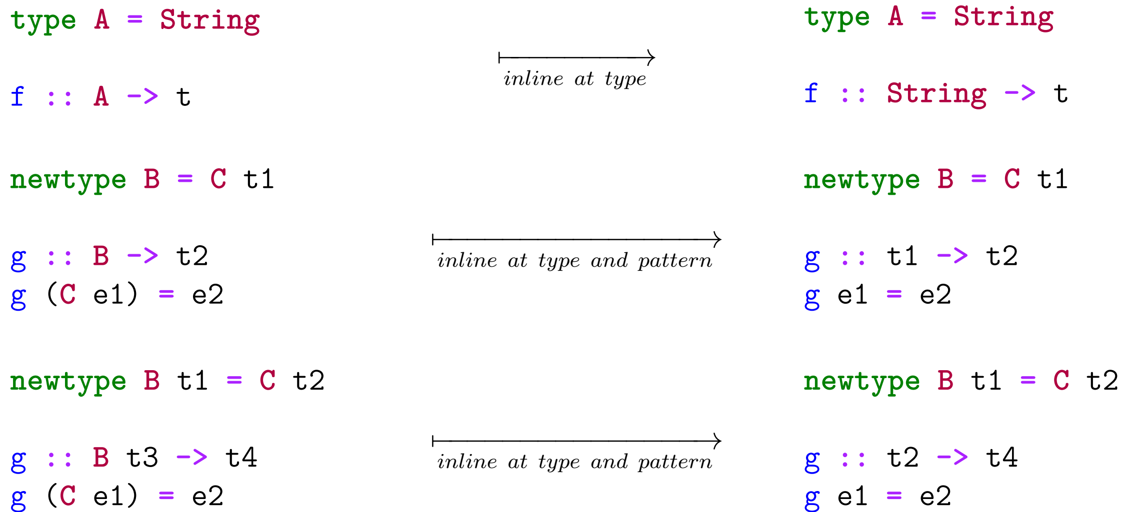

Figure 3.11.: Inlining data types.

Template Haskell

hsreduce also tries to dump Template Haskell splices. This pass is only possible if the test case succeeds the renaming and typechecking phases of GHC. If we succeeded in obtaining a renamed source of the test case, we write it out to the temporary file and check if the test-case is still interesting. This has the effect of expanding all splices at once. In contrast to dumping single splices one by one, this approach does not work for test-cases that fail during typechecking. For future work, we also want to support dumping single splices by running GHC with the `-ddump-splices` option and for each splice, try to dump it.

Pass 28: Expand Template Haskell Try to expand all Template Haskell splices at once.

Inlining Data Types

Additionally, we try to inline type aliases and newtypes. This pass needs two things to happen: inlining at types and inlining at patterns. Examples can be seen in figure 3.11.

Pass 29: Inline Data Types For prefix data constructors and newtypes, try to inline them in type signatures and patterns.

Type Classes

For type classes, we try to remove unused methods, both from the classes and instances. See figure 3.12.

Pass 30: Remove Typeclass Methods For each method, try to remove it simultaneously from the typeclass and all its instances.

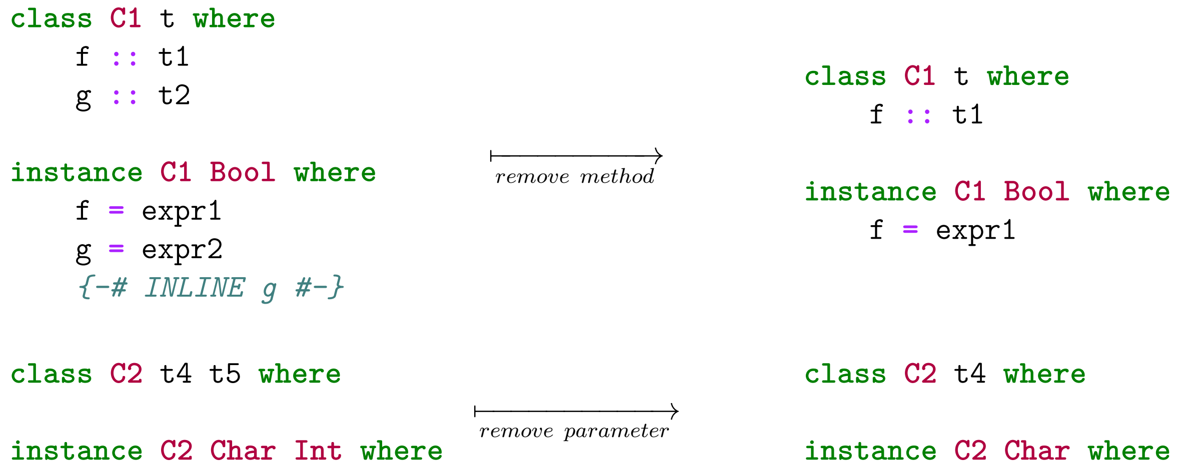

Figure 3.12.: Handling typeclasses.

Inlining Functions

Before inlining we look at functions with only one match and try to eta-reduce them.

First, we look at trivial inlining opportunities, where we have a binding of the form $f = g$, so we just replace occurences of f with g.

For other function bindings, hsreduce tries to do true inlining. There we look for occurences of the function name and replace it with a lambda expression. Then, we try to do beta reduction on the lambda application expression. For functions with multiple right-hand sides or multiple matches, there is not much we can do, because we might not know the value or the "form" of the argument.

So for inlining, the question when to inline is important. Inlining too many functions leads to unreducible large lambda expressions that decrease readability of the test-case.

We have chosen to inline when there is only one usage site and if the function has got one match, one right-hand side and no guards.

Pass 31: Eta-reduce matches For each match, try to eta-reduce it.

Pass 32: Inline Function If a function has only one match and only one usage site, try to inline it as a lambda function.

Pass 33: Beta-reduce Expressions For expressions of an expression being applied to a lambda, try to beta-reduce it.

One could also merge these passes into one pass but separating them might lead to more flexibility.

Type Families

For type families, hsreduce sees if it can apply them. We look at all type family equations and occurences of the type family and see if one of the equations can

```
type family F a where
    F Int = Char
    F a = String

f :: F Int -> ()

g :: F Bool -> Bool
```

$\xrightarrow[\textit{reduce applications of } \mathsf{F}]{}$

```
type family F a where
    F Int = Char
    F a = String

f :: Char -> ()

g :: String -> Bool
```

Figure 3.13.: Handling type families.

be applied. The best way to do it would be to use information provided by the typechecker of GHC and to use it to normalize types with type family applications in it. However, when trying out those type normalizing functions, they failed to reduce type family applications that could already be reduced by syntactic transformations. It seems that we do not apply those type normalizing functions correctly and we did not find a way to do so. We currently only do syntactic transformations to reduce type family applications. Our approach also lacks pattern matching, so the transformations we can do are somewhat limited.

Lastly, we try to remove type family equations.

See figure 3.13 for an example.

Pass 34: Reduce Type Family Application For application of a type family, try to reduce it using one of the type families equations.

4. Merger Implementation

While we now have the means to reduce single Haskell modules, most bugs come up in large projects, consisting of multiple inter-dependent Haskell modules. We need to squash these down to a single module that our reduction machinery can boil down. Haskell projects can be handled by several tools, cabal [9] and stack [10] being the most popular choices, but make [11] and shake [12] can also be used. In this work, we will focus on cabal projects. Because stack also uses cabal files, this approach should also work for stack projects. With that, our work should cover the two most popular build tools for Haskell.

Cabal projects are specified using cabal files which specify the name of the project, its dependencies, its build artefacts (libraries or executables) and their constituents, flags that should be passed to the compiler and more.

4.1. Preprocessing

First, one often needs to simplify the cabal file. This includes removing unnecessary sections like unused benchmarks, unused testsuites, unused libraries and executables and unused modules. Another preprocessing step is to inline dependencies. These steps are not automated yet because with the current test-cases it was quicker to do it by hand so far. On the contrary, merging modules most often cannot be done by hand because the number of names can quickly increase, even with a small number of modules. Therefore, we need to automate this step.

4.2. Existing Tools for Merging

First, we looked if there are already existing tools we could use to do merging of Haskell modules. *hs-all-in-one* [13] is a Haskell project by Joachim Breitner. It uses the *haskell-src-exts* [14] library for parsing and works on its AST representation. This library is very user-friendly and the implementation is quite small, however, it does not work for all projects. One example: we could not merge files that use the C preprocessor language extension, although that is solvable by using the *hse-cpp* [15] library. Additionally, *hs-all-in-one* only disambiguates names from the modules we want to merge. This can lead to ambiguous names from conflicting imports. Problems we encountered when trying to use *hs-all-in-one*:

- not in scope data types
- ambiguous occurrences (data types, operators not being renamed)

```
{-# language CPP #-}

#if __GLASGOW_HASKELL__ > 710
main = pure ()
#else
main = undefined
#endif
```

(a) Example using the CPP extension.

```
import Data.List
import A

main = pure ()

f = map id [1,2,3]
```

```
{-# language NoImplicitPrelude #-}
module A where

import qualified Prelude
import Data.Map

g = map Prelude.id Prelude.$ fromList [(1, 'a')]
```

(b) Ambiguous occurence of map when merged.

Figure 4.1.: Two examples that hs-all-in-one cannot merge.

- multiple declarations for the same name
- duplicate type signatures
- names where the module name did not fit with the import module name
- external names not being qualified
- operators not being renamed and clashing

Finally, we learned that the *haskell-src-exts* library is not maintained anymore [16]. This will in the future lead to a mismatch between what the GHC API can parse and what this library can parse. These issues led us to write our own version of a merging tool using the GHC API. The goal was to write a merging tool that is more future-proof (by using GHC's parsing infrastructure) and works for a wider range of Haskell programs and projects.

4.3. Requirements for Merging Haskell Modules

First, we need to get the build information from the project files, namely being cabal files. This is information such as the modules that are to be built, as well as the flags and extensions to set and enable, respectively. Especially, we need to obtain the module graph. Mostly, this is information to correctly set up a GHC API session.

Next, each module needs to be parsed and renamed. The renamer resolves for each name its provenance: where it is declared and from where it is imported [17].

Third, the information from the renamer needs to be applied on names occuring in the parsed or renamed source.

Lastly, all the renamed modules need to be merged into one big module.

4.4. Obtaining Project Information

We found three ways to get the build information:

the first is to use the Cabal library to parse the cabal file. By doing that, a package description is obtained, which contains the different components declared in the cabal file. Disadvantageous here is that the users have to setup a GHC session themselves. That means they have to fetch all the correct flags, include directories and source dirctories themselves.

Secondly, GHC plugins [18] can be used. Here, users would build the project with cabal, but with a flag specifying their plugin. The plugin then automatically provides the build information and the right flags and no manual management of the GHC API session is needed. But the disadvantage is that the plugin is only run on modules individually. Using this tool would require to first rename the modules, print that information in temporary files and then run a separate tool to merge the renamed modules.

Lastly, there is the *hie-bios* [19] library by Matthew Pickering. It is used by the *haskell-language-server* [20] and *ghcide* [21] project to set up GHC API sessions [19]. It supports [19] a wide range of Haskell packaging tools including *cabal-install* [9], *stack* [10], *rules_haskell*, *hadrian* and *obelisk*. We opted for using hie-bios, because it offers the easiest way to setup a GHC API session. For users of our merging utility, this means they have to create a hie.yaml file and write down the name of the executable or library to which the error inducing module belongs to. Then they have to specify the name of the latter module, Our tool finds the hie.yaml file and takes it from there.

4.5. Accessing Renaming Information

After the renaming phase, GHC should know for each name the module from where it was declared and imported. Now we need a way to access this information to use it for our merging tool. We found two major ways to get module information about a name: nameModule_maybe and lookupGRE_name.

The first lookup function we used was nameModule_maybe, which given a name, returns the module where that name was declared. This leads to the problem that the name of the declaring module might be very different from the name of the importing module, as can see in example figure 4.2. Very often they are so different that it is impossible to find out the importing module name from the declaring module name. In the example that is the case, we have no way of knowing that Prelude re-exports things that were declared in GHC.Types.

```haskell
main = pure ()

n :: Int
n = 3
```

(a) Original file.

```haskell
import qualified Prelude

main = GHC.Base.pure ()

n_Main :: GHC.Types.Int
n_Main = 3
```

(b) Using nameModule_Maybe

```haskell
import qualified Prelude

main = Prelude.pure ()

n_Main :: Prelude.Int
n_Main = 3
```

(c) Using lookupGRE_name

Figure 4.2.: Observable differences in using two provenance functions.

lookupGRE_name uses the global renamer environment table. Given a name, we can find its corresponing element in that environment and its provenance, where we find exactly the module which brought this name into scope.

We chose to use lookupGRE_name since it gives us exactly what we want: the importing module name for external names.

4.6. Essential Renaming

Now that we have a way to find the correct module information for a name, at which places should we apply it? Should we apply it at every name that we encounter? Unfortunately, it does not suffice to just rename all names we encounter. Let us first look what we should do for names that are interesting for us.

4.6.1. Mapping of Names

A name we encounter can only be one of three things: a built-in name (something like Haskell list syntax), an external name (brought into scope by an import) or an internal name (a name from a module of the module graph we are currently working on). For built-in names, we do not have to do anything. External names we qualify to be explicit where this name is coming from. Internal names we disambiguate by concatenating the module name with the name.

4.6.2. Additional Renaming

Record fields are not renamed by our name-renaming function, since they only contain RdrNames. Some of them are not disambiguated until type checking because the user might have enable overloaded record fields. So we have a separate function to properly rename them.

4.6.3. Prohibited Renaming

Our name-renaming function qualifies some things that should not be qualified:

- binding positions (for example type class method names)
- type signatures
- names of type families

We run separate functions to unqualify them.

4.7. Applying Renamer Information

Now, there are two major ways to apply the provenance information that we got from the renamer: we can either apply it on the parsed source or the renamed sourced.

4.7.1. On Renamed Source

With the renamed source, we can directly change the names there. We also do not have to do any clean up after merging, which is necessary with other approaches. Another advantage is that we always get one module, we never have to exclude modules from the merging process.

But we still have problems: Template Haskell splices with hidden names being expanded cannot be handled by this approach because we cannot get provenance information for those names except for their exact module name where they were declared. There are two problems that can appear: either the exact module name is a hidden module or the name itself is a hidden name. In figure 4.4 there is an example for that. In the merged file, there is `valueConName` being used, which is a non-exported, hidden name and additionally there are hidden modules like `Data.Aeson.Types.FromJSON` being used. To our knowledge, this problem appears only in a small amount of cases and libraries that use Template Haskell seldom expand their splices to expressions using hidden symbols. One solution would be for libraries to change their Template Haskell code to not expand with hidden names.

Because of these problems we briefly considered using the parsed source, but there we encounter the problems of having to indirectly apply the renaming information and running into the Template Haskell staging restriction.

4.7.2. On Parsed Source

Applying the changes on the parsed source means for each name in the renamed source that its location and its corresponding change are to be saved. It has the advantage that template haskell splices possibly containing hidden symbols are not expanded.

But there was then the problem, that splices were not renamed properly. Somehow, their locations are not visited during the renamer. So in another run, splices are focussed explicitly and their names are renamed.

But then shows up another problem that there might be functions in splices which are declared in the same module which violates the staging restriction. This can be solved by excluding the module where that function is declared from the set of modules to merge.

So, using the parsed source, in some cases it might not be possible to merge all modules into one.

4.7.3. Re-Exports by "Our" Modules

Another problem that can appear are re-exports by modules that belong to the set of modules we want to merge. In figure 4.5 there is an example where we would resolve for `map` its declaring name. This came from us choosing the declaring name over the importing name if the two differed. The rationale for this was, if we import one of "our" modules (the modules belonging to the graph we want to merge), which is exporting a name that was declared in another one of our modules, then we need to choose the declaring name. In this case, we resolved the importing name for `map` to be module `A`, which is correct, but it is not *really* the module where `map` is coming from. To find that out, we need to check, whether module `A` is declaring or importing `map`, and if that is the case, follow up recursively. So we included a check: if the importing module name belongs to "our" modules but the declaring module name differs, then we recursively query for the importing module name.

```haskell
{-# LANGUAGE TemplateHaskell #-}

import A

main = $(f)
```

(a) Main.hs

```haskell
{-# LANGUAGE TemplateHaskell #-}

module A where

import Language.Haskell.TH

f = varE $ mkName "undefined"
```

(b) A.hs

(c) Initial files

```haskell
{-# LANGUAGE TemplateHaskell #-}

module AllInOne where

import Language.Haskell.TH

f_A = varE $ mkName "undefined"

main = $(f_A)
```

(d) Result using parsed source

```haskell
{-# LANGUAGE TemplateHaskell #-}

import qualified Prelude
import qualified Language.Haskell.TH

f_A
  = Language.Haskell.TH.varE
      Prelude.$ Language.Haskell.TH.mkName "undefined"
main = (Prelude.undefined)
```

(e) Result using renamed source

Figure 4.3.: Staging restriction.

```
{-# LANGUAGE TemplateHaskell #-}
import Data.Aeson.TH

main = pure ()

data A = B

$(deriveJSON defaultOptions ''A)
```

(a) Initial file

```
{-# LANGUAGE TemplateHaskell #-}
module Main where
import qualified Prelude
import qualified Data.Aeson.TH
main = Prelude.pure ()

data A_Main = B_Main

instance Data.Aeson.Types.FromJSON.FromJSON A_Main where
  parseJSON
    = \ value_a8m9
        -> case value_a8m9 of
             Data.Aeson.Types.Internal.Array arr_a8ma
               | Data.Vector.null arr_a8ma -> Prelude.pure B_Main
               | Prelude.otherwise
               -> (((Data.Aeson.TH.parseTypeMismatch' "B") "Main.A")
                     "an empty Array")
                    ("Array of length "
                       Prelude.++
                         (Prelude.show Prelude.. Data.Vector.length) arr_a8ma)
             other_a8mb
               -> (((Data.Aeson.TH.parseTypeMismatch' "B") "Main.A") "Array")
                    (Data.Aeson.TH.valueConName other_a8mb)
instance Data.Aeson.Types.ToJSON.ToJSON A_Main where
  toJSON
    = \ value_a8m7
        -> case value_a8m7 of {
             B_Main -> Data.Aeson.Types.Internal.Array Data.Vector.empty }
  toEncoding
    = \ value_a8m8
        -> case value_a8m8 of {
             B_Main -> Data.Aeson.Encoding.Internal.emptyArray_ }
```

(b) merged file

Figure 4.4.: Problems caused by Template Haskell splices.

```
import A

main = pure ()

a = A.map id [1,2,3]
```

(a) Main.hs

```
module A (map) where

import B
```

(b) A.hs

```
module B (map) where

import Data.List
```

(c) B.hs

```
module Main where
import qualified Prelude
import qualified Data.List


main = Prelude.pure ()
a_Main = GHC.Base.map Prelude.id [1, 2, 3]
```

(d) Merged file with problem with our previous approach

Figure 4.5.: Re-Export by "our" modules.

5. Evaluation

To evaluate our new reducer, we evaluated it on some real-world issues from the GHC tracker and measured the reduction quality measured in bytes, tokens and identifiers, the time spent and the total test invocations. We compare it to the best of the related work, C-Reduce. Finally, we look at the passes individually and the pass ordering.

We created a separate repository and added 11 test cases that were derived from real-world issues directly from the GHC tracker [1]. They exhibit a variety from bugs ranging from the issues in type checker, the simplifier, to bugs that show up in decreased time and space performance. Another variety that the test cases exhibit is that for each there are different techniques needed to reduce it. As examples: for T14270 turning expressions into *undefined* does most of the work, for T13877 we needed to turn types into the *Unit* type and for T16979 we need to reduce type family applications. Almost all these issues are triggered by different GHC versions [2]. With T15696_2 and T18140_2, the Haskell files were produced by using our merging functionality.

Evaluation was done on a Lenovo Yoga 920 with an Intel i7 eight core CPU and 16 gigabytes of RAM. We evaluated with C-Reduce version 2.9. On all test cases we called C-Reduce and hsreduce with eight interestingness tests running concurrently and having a timeout of 30 seconds [3].

5.1. Comparison with C-Reduce

We compare with C-Reduce because it is the best one of the related work. Our goal is to see how many identifiers and bytes it can reduce, in what amount of time and with how many test invocations. We focus on reduction of identifiers because it is a stronger metric than reduced number of bytes since C-Reduce shortens identifiers.

5.1.1. Results

Our tool is better at reducing identifiers than C-Reduce. hsreduce removes in the median 78 percent of the identifiers compared to C-Reduce being able to remove

[1] See appendix for links to the issues

[2] We used the nix package manager to get specific GHC versions and in one test case we build a specific commit of GHC

[3] On T18140_1 we called C-Reduce and hsreduce with two threads and a timeout of 150 seconds, on and T18140_2, we called C-Reduce and hsreduce with two threads and a timeout of 150 seconds

60 percent in the median. Looking at the geometric mean, hsreduce removes 79 percent of identifiers compared to C-Reduce being able to remove 70 percent. This suggests that hsreduce's transformations are more meaningfull. We think hsreduce's domain-specificity seems to lead to that that difference.

T16979, T18098 and T13877 are test cases where hsreduce performs much better than C-Reduce. With T16979 hsreduce achieves a 60 percent smaller end result while only spending one third the time. With T18098 hsreduce achieves a 30 percent smaller end result while spending 1/17 the time. For T13877 hsreduce achieves a 50 percent smaller end result while spending less than half as much time. These test cases profit from domain knowledge which hsreduce is able to exploit. T8763 is a test case where hsreduce is surprisingly bad compared to C-Reduce. hsreduce only spends half as much time but its end result is 39 percent bigger. Reasons for this are still unclear.

For all other test cases both tools are close regarding end reduction size, but hsreduce is about two times faster. For these tests, domain-specificity is not needed as much for the previously mentioned ones. Another reason is that C-Reduce also does shortening of identifiers, which improves the reduction in bytes but decreases readability.

Looking at the reduction in identifiers, hsreduce is in all but one test case the better reducer. This suggests that the reductions of hsreduce are more meaningful.

hsreduce does about a tenth the interestingness test shell script invocations that C-Reduce does.

The results of the comparison can be seen in table 5.1. For each test case we recorded:

- the number of identifiers before reducing, after reducing with C-Reduce and after reducing with hsreduce
- the time spent by both tools
- the amount of test invocations done by both tools
- the number of bytes before reducing, after reducing with C-Reduce and after reducing with hsreduce

For the column *Identifiers* there are three sub-columns: the column "s" (short for starting value) stands for the number of identifiers of the test case before any reduction. For test case T8763 that is 221 identifiers. The column "C" (short for C-Reduce) stands for the number of identifiers in the result of C-Reduce's test case reduction. For T8763 there were 63 identifiers left after running C-Reduce on it. In parentheses we have the reduction in percent. For T8763, C-Reduce resulted in a -72% change in the number of identifiers. To calculate this value, we take the ratio of the number of identifiers after reduction compared to the number of identifiers before the reduction. In this example that is $\frac{63}{221} = 0.285$. Now we subtract this from one and get $0.715 = 71.5\%$. Because this represents the amount of identifiers that were reduced away, we negate it. This is how we get at at the rounded value of -72% in the table. Because for C-Reduce this value it achieved on T8763 is higher than the one achieved by hsreduce, we put it in boldface. For the "winning" tool

regarding a certain test case, we always mark this value in boldface. The next column in the *Identifiers* multi-column is "h" (short for hsreduce), which stands for the number of identifiers in the result of hsreduce's reduction of the test case. For T8763, there were 84 identifiers left after running hsreduce and hsreduce resulted in a -62% change in the number of identifiers, which is calculated the same way like it was for C-Reduce. As the last column of the *Identifiers* multi-column, we have the column "f" standing for a factor, where that factor tells us how much bigger the amount of identifiers in the result of the losing tool is, in comparison to the amount in the result of the winning tool. Here with T8763, hsreduce is the worse performing tool. So we calculate the number of identifiers for the result of hsreduce and divide that by the number of identifiers in the result of C-Reduce. For T8763, we calculate $\frac{84}{63} = 1.33$.

For *Time Spent*, there are only two columns: one for how much time C-Reduce spent and one for how much time hsreduce spent. We display the time spent in seconds.

Tests Run displays for both tools, how many interestingness test shell script invocations they did.

The last multi-column is about *Bytes*, meaning the byte size of the test case before and after reduction. Like with *Identifiers*, we have a starting value, a value for C-Reduce and one for hsreduce. Calculation of these values is analogous to the ones in *Identifiers*, except for that we look at the byte sizes instead of the number of names.

Lastly, we include median and geometric mean for the reductions of C-Reduce and hsreduce. These are calculated like this: for the median, we take the ratios of the amount of identifiers after reduction and the amount before reduction, which we calculated earlier ($\frac{63}{221}$ for C-Reduce's result on T8763) for each test case and calculate the median for these gathered ratios. We then subtract this value from one, turn it into a percentage and negate it. For the geometric mean, the calculation is analogous.

Test Case	*Identifiers*				*Time Spent*		*Tests Run*		*Bytes*	
	s	C-Reduce	hsreduce	f	C	h	C	h	C-Reduce	hsreduce
T8763	221	63 (**-72%**)	84 (-62%)	1.33	2624	1310	13640	1067	689 (**-73%**)	957 (-63%)
T13877	283	119 (-58%)	45 (**-84%**)	2.64	840	313	18425	1424	1057 (-59%)	560 (**-78%**)
T14270	372	41 (-89%)	36 (**-90%**)	1.14	393	76	12070	794	389 (**-89%**)	439 (-88%)
T14779	139	59 (-58%)	55 (**-60%**)	1.07	240	70	4546	762	521 (**-78%**)	561 (-76%)
T14827	66	48 (-27%)	46 (**-30%**)	1.04	4560	2744	124	1041	2397 (**-88%**)	2533 (-87%)
T15696_1	165	88 (-47%)	65 (**-61%**)	1.35	240	161	3323	512	431 (**-69%**)	436 (-68%)
T15696_2	4297	215 (-95%)	127 (**-97%**)	1.69	4980	4675	43339	4345	1968 (-96%)	1477 (**-97%**)
T16979	1164	661 (-43%)	230 (**-80%**)	2.87	3554	1276	56023	6177	5219 (-31%)	2026 (**-73%**)
T18098	1361	516 (-62%)	315 (**-77%**)	1.64	63000	2728	194528	8678	3668 (-68%)	2459 (**-75%**)
T18140_1	628	118 (-81%)	98 (**-84%**)	1.20	24000	21531	27098	3594	4204 (-76%)	3704 (**-79%**)
T18140_2	1634	-	111 (**-93%**)	-	-	33691	-	6379	-	4783 (**-94%**)
median		-60%	**-78%**						-74%	**-77%**
geom. mean		-70%	**-79%**						-78%	**-81%**

Table 5.1.: Comparison C-Reduce and hsreduce; abbreviations: s (starting value), C (C-Reduce), h. (hsreduce) and f (size increase factor); time spent in seconds, geometric mean (geom. mean)

5.1.2. Looking at Test Cases

Here are some noteworthy observations we came across while evaluating.

T16979 is the test case where hsreduce does best in comparison with C-Reduce. It manages to reduce to a size, that a human expert got in their second version of reduction. To get to this result, hsreduce does several things.

Reduce Type Family Applications The main obstacle of this test case are its many type families. They are declarations which we can remove but first their equations need to be unused. At the beginning, this is not the case. All type families are referenced in types. To free those references, we use our pass to reduce type family applications. That enables our pass to remove type family equations, which might free more identifiers referencing other type families. At last, we can remove the whole declaration. Even though we only reduce type families syntactically, we can remove all type families in that test case. To be able to remove them, one needs to reduce their applications.

Remove Typeclass Profunctor To be able to remove this declaration, references to its typeclass method #. need to be freed. This happens by turning undefined #. I undefined into I undefined at some point. This was done by the pass that reduces expressions to their subexpressions. Next, we can remove the typeclass method, the instance and lastly remove the typeclass declaration.

Remove Typeclass GenericN Another typeclass that is removed is GenericN For that to happen, the references to GenericN, toN, fromN need to be removed. GenericN gets freed by removing it from a context in another typeclass instance. fromN gets freed by turning fromN undefined into just undefined toN also gets freed by turning a reference to it into undefined. After turning those three all dead, we can remove its instance and typeclass declaration.

Remove Instances We additionally remove instance for GHasParam and GHasParamRec. These removals are enabled by removing their identifiers from contexts and turning references to them in types to the unit type.

Remove Functor and Applicative Instances There are also Functor and Applicative instances for the Yoneda type which can be removed. For Applicative, this is enabled by turning a use of <*> into undefined

Inline Functions Lastly, there are several one-match-one-call-site functions that can be inlined.

Of all the previously mentioned changes, removing the Profunctor typeclass and its instances are the only change which can also be done by C-Reduce.

T18098 is another ticket where hsreduce achieves better performance than C-Reduce. C-Reduce spent 17 hours producing the result. For C-Reduce, this is the slowest test case. The running time is three times longer than the next worst test

case. We do not know why C-Reduce's running time was so high for this test case, it also has a high amount of test invocations, almost four times as much as the next worst one. It seems that its passes are not as effective in this test case as they are in others. In comparison, hsreduce took less than one hour, while having the result being 30 percent smaller C-Reduce's result still has 30 function bindings, while hsreduce's result has 16.

Inlining Functions This is because right as one of the first reduction steps hsreduce inlines more than five functions. For example it replaces nunstream with (\s -> s ‘seq‘ New (gmvunstream s)). After inlining those functions their declarations aren't referenced anymore and can be removed later.

Removing Parameters Next, hsreduce removes parameters from functions, for example simultaneously turning the function application expression checkedAdd Exact m n into checkedAdd m n, turning the type signature checkedAdd :: (Int -> Size) -> Int -> Int -> Size into checkedAdd :: Int -> Int -> Size and turning the match checkedAdd con m n into checkedAdd m n.

Removing Constructor Arguments hsreduce also removes arguments from the Bundle data constructor. It does that simultaneously in function patterns and expressions.

C-Reduce cannot do those transformations. After those transformations are done, a lot can be removed by removing matches, signatures and declarations.

T8763 exhibits a performance bug in GHC where a loop written with a library function allocates 50 percent more than a hand-written loop. It is the worst test case for hsreduce in comparison with C-Reduce. C-Reduce beats hsreduce by 39 percent lower reduction size. hsreduce fails to perform reductions like turning `[3, 5 .. n]` into `[]`, `\k -> do unsafeWrite sieve k p` into `\k -> k` or `when isPrime $ m` into `m`. These are all reductions that hsreduce normally can do. When trying one of these by hand on hsreduce's result, we saw that the test case becomes temporarily uninteresting. Uninteresting here means that it the difference became too big, for example reducing too aggressively might turn the hand-written loop into a no-op. However, applying several of these leads to the test case being interesting again. We do not yet know, how C-Reduce does these changes and how it bridges that gap of this uninteresting test invocation.

For all other test cases, the results between C-Reduce and hsreduce were surprisingly close. Even though these test cases use several language extensions, C-Reduce still reduced more than 60 percent in those. Explanations for this are that these files have advantageous formatting and advantageous structure, where line-based reductions and reductions of braces, brackets, and parentheses with their contents still work well.

5.1.3. Discussion

An advantage of C-Reduce is that it can apply its deletions in a very general way. C-Reduce manages through its language-agnostic passes such as line deletion, token removal, removing an operator and one of its operands, removing balanced pair of curly braces and all the including text, and renaming tokens to capture about the same reduction quality that we get with our more trivial passes. For example, removing declarations is something that C-Reduce also manages to do by trying to delete a certain amount of lines. C-Reduce is better tested, optimized and language agnostic, so it might also be better prepared for new language extensions coming to the Haskell language.

In general, it is remarkable how effective C-Reduce is, given that it lacks domain knowledge of Haskell. But hsreduce is able to match that performance and even beat C-Reduce in some cases, especially when there are more advanced features like type families and functions with multiple parameters. Also, looking at the number of identifiers, hsreduce's reduction seems to be better. This is owing to Haskell domain knowledge. Here are passes that none of the related work can do:

- using the undefined expression, Unit type, wildcard type expressions and wildcard patterns
- removing unused data constructors
- turning record constructors into prefix constructors
- replacing a case- or an if-expression by one of their branches
- inlining functions
- reducing type family application
- removing function parameters simultaneously from declaration and call sites
- deleting typeclass parameters simultaneously from typeclass and instances
- deleting methods simultaneously from typeclass and instances

Another advantage of hsreduce is that does not use the byte size as a fixpoint criteria. For example turning a short variable name into `undefined`. At first glance this makes the test case worse because its size increases. But this might enable other passes later and lead to a lower byte size in the end.

Lastly, none of the related works can merge modules in any way. They only work on single Haskell files with an interestingness test. So for test cases like T15696 and T18140, where with hsreduce one can get to results of same quality achieved by human experts, one would not even get to the starting point of the reduction (the merged module) with the other tools.

Our tool is still far from perfect. Especially on T8763 and T14827 we think there is still a little bit more reduction that could be done. Also it seems that hsreduce cannot exploit its low number of test invocations. Looking at the time spent, the difference to C-Reduce is not as large as it is with the test invocations. hsreduce makes about 1/16 test invocations compared to C-Reduce but only manages to spent 1/2 of time compared to C-Reduce. There still seems plenty of overhead or inefficiencies in the parallelization that can be worked on. Also confusing is that in T14827, hsreduce does more test invocations than C-Reduce but still arrives at a similar running time. My suspicion is that the shorter the running time of the interestingness test and the larger the AST, the cost of traversing the AST becomes larger. This might be solvable by creating hand-written instances of the generic traversal for the GHC AST instead of using the derived ones, which might make traversal faster.

5.2. Comparison with other Related Work

Compared to ddmin [2], our tool produces only syntactically valid changes. Similarly to HDD [6] and Perses [8] it also uses structural information to make changes, for instance reducing to subexpressions. They do not support Haskell yet, but even if they did, hsreduce works for a much wider variety of Haskell programs, since GHC has many extensions which are not in the Haskell 2010 standard. structureshrink [3] also uses structure to guide reduction but is five to ten times slower and fails to produce reproducible examples in multiple test cases. Also similarly to halfempty [4] and C-Reduce [1] it uses parallel test case execution assuming that most of them will fail to speed up the reduction process. And like C-Reduce it employs a modular reducer infrastructure with a pluggable set of passes.

5.3. Merging

To evaluate our merging functionality we used two test cases: T15696_3 and T18140_3. With T15696_3 we can merge it into a 56 kilobyte file and with T18140_3 we can merge it into a 800 kilobyte file. With both test cases, the merged files are still interesting. The merged file for T15696_3 is very similar to the test case of T15696_2, which resulted from using an earlier version of our merging utility, so it was not evaluated in section section 5.1. The merged file for T18140_3 was not evaluated because that level of merging functionality was reached at a very late stage of the work, so it was not evaluated due to time constraints.

5.4. Evaluation of Passes

Because our search is greedy and might only reach a local optimum regarding the reduced number of identifiers and because it seems that some passes inhibit other

passes, it is important in which way we order the passes. After that, we also look at pass statistics to see, which passes were worth implementing and which ones were not.

5.4.1. Pass Ordering

We also looked at what effect different ordering of the passes has on reduction performance. To evaluate this, we ran hsreduce on the test cases T13877, T14270, T14779, T16979, T18098 and T8763 with a shuffled ordering and collected performance statistics. First, we looked at putting all pure passes into one list and running those. We tried out 164 different orderings. We summed up the end number of bytes, tokens, identifiers and running times.

The reduction ranged from 518 identifiers for the best ordering and 850 identifiers for the worst one. Looking at byte sizes, it ranged from 5268 bytes to 7299 bytes. So the right ordering can improve reduction by up to 27.8 percent.

Looking at running times, the fastest ordering took 2317 seconds and the slowest one took 3477 seconds. If one is interested in speedy reduction, choosing the right ordering can speed up things up to 33.3 percent. Another observation is that reduction quality and running time seem to be opposing goals. Here the correlation between end name size and time spent is -0.1879.

We chose to use pass ordering nr. 1, which lead to 30 percent improvements in reduction performance with T16979 and T18098. Pass ordering nr. 1 can be seen in figure 5.1a.

We have not gotten any explanations yet on why one ordering is good or bad. One example we saw, was that putting *type2Wildcard* before *type2Unit* in a previously used ordering decreased performance considerably. This seems to be a weakness of the search algorithm, since taking the first change possible might inhibit better changes

5.4.2. Pass Statistics

In table 5.3 we look at pass statistics to see which passes were how successful. This is to guide future efforts so that others can see which areas might be more fruitful to write passes for.

So what makes a pass valuable? A valuable pass would have a low amount of total invocations, a high ratio of successful to total invocations and enabling a lot of other passes, mainly by deleting a large amount of identifiers. To delete a lot of identifiers, it should be applicable to large elements and to a large number of elements, but this would also lead to a high number of test invocations. We expect that a high amount of deleted identifiers correlates with a high number of test invocations. To get a high rate of success, the pass should focus on a specific type of element, for example only handle expressions with a certain form and do nothing for the rest.

We defined two metrics, rate of success that is, the ratio of successful to total invocations and efficiency the number of removed identifiers divided by the total

ordering nr.	bytes	tokens	identifiers	t. s.	s. i.	t. i.
1	5268	1215	518	3050	862	22363
2	6024	1467	680	2820	841	15996
3	6284	1505	689	3182	801	21350
4	6318	1520	698	2620	781	17398
5	6284	1530	700	3009	833	21285
6	6260	1509	700	2963	785	13462
7	6318	1514	701	2729	831	19159
8	6261	1514	701	2663	786	12469
9	6249	1523	705	2345	771	22882
10	6408	1521	707	3133	744	22146
11	6514	1556	717	3138	812	14958
12	6472	1563	721	2770	855	18479
13	6379	1563	721	2559	795	20301
14	6466	1577	723	3120	890	17784
15	6527	1586	729	3073	814	14177
16	6510	1574	732	2891	760	13961
17	6552	1593	732	3163	819	15998
26	6578	1612	744	2635	763	19986
27	6612	1628	747	2623	841	17827
28	6695	1646	748	2601	795	23076
29	6925	1683	764	3477	827	26149
30	6710	1688	764	2541	763	25666
31	6644	1644	767	2720	834	20481
32	6856	1658	767	3403	842	22991
33	6784	1659	767	3535	807	14203
34	6651	1653	774	2526	857	22596
35	6654	1655	774	2818	814	21358
36	6934	1704	780	2641	856	15635
37	6634	1674	781	2901	779	26792
38	6607	1658	782	2956	899	20131
39	7021	1733	782	2744	821	22020
40	6806	1713	783	3297	835	21839
41	6968	1714	784	2604	796	21082
42	6967	1705	785	2698	813	28935
43	6818	1718	787	2778	879	20998
44	6752	1697	793	2549	699	18667
45	7078	1719	797	2430	802	18842
46	6960	1740	800	2918	836	22316
47	6954	1710	808	3037	763	22933
48	7299	1840	850	2999	867	23766

Table 5.2.: Comparison of different pass orderings; abbreviations: time spent (t. s.), successful invocations (s. i.), total invocations (t. i.)

```
[ rmvRHSs
, TypeFamilies.apply
, TypeFamilies.rmvUnusedParams
, pat2Wildcard
, rmvFunDeps
, localBinds
, betaReduceExprs
, TypeFamilies.familyResultSig
, Functions.inline
, Parameters.rmvUnusedParams
, rmvConstructors
, unqualImport
, rmvDerivingClause
, DataTypes.inline
, simplifyType
, TypeFamilies.rmvEquations
, rmvConArgs
, rmvGuards
, rmvMatches
, simplifyDerivingClause
, tyVarBndr
, contexts
, type2WildCard
, rmvImports
, filterExprSubList
, etaReduceMatches
, rmvSigs
, handleMultiParams
, splitSigs
, rmvTyClMethods
, type2Unit
, expr2Undefined
, simplifyConDecl
, Typeclasses.rmvUnusedParams
, rmvDecls
, simplifyExpr]
```

(a) Best ordering

```
[ handleMultiParams
, simplifyType
, rmvMatches
, TypeFamilies.rmvUnusedParams
, rmvGuards
, betaReduceExprs
, type2WildCard
, etaReduceMatches
, simplifyDerivingClause
, rmvDerivingClause
, rmvConstructors
, splitSigs
, TypeFamilies.familyResultSig
, pat2Wildcard
, Functions.rmvUnusedParams
, rmvImports
, tyVarBndr
, filterExprSubList
, inlineType
, contexts
, rmvRHSs
, simplifyConDecl
, unqualImport
, rmvConArgs
, rmvDecls
, simplifyExpr
, rmvFunDeps
, localBinds
, rmvTyClMethods
, TypeFamilies.apply
, Typeclasses.rmvUnusedParams
, rmvSigs
, Functions:inline
, type2Unit
, expr2Undefined
, TypeFamilies.rmvEquations
]
```

(b) Worst ordering

invocations.

To calculate them, we record for every pass the number of: successful invocations, total invocations, removed bytes, removed tokens and removed identifiers. These were recorded with 32 different pass orderings to lessen the influence of the pass ordering. The results can be seen in table 5.3.

The best passes, based on identifiers removed were: removing declarations, turning types into the unit type, removing signatures and removing matches. The worst passes were inlining functions, splitting signatures, removing constructor arguments and inlining types. This makes sense, since declarations are one of the largest elements that can be deleted at once and for inlining functions it also makes sense, since it does not remove anything but transform a function call into a lambda which at first seems to make the test case “worse”. Turning types into the unit type and removing matches seem to remove a lot of identifiers because those elements come up very often in programs.

For rate of success, passes that work on types that appear a lot in Haskell programs were the worst. Examples of these passes are turning types to the unit type, turning expressions into `undefined`. The passes with the highest rate of success were spezialized passes focussing only on certain types of elements in the GHC AST like beta reducing expressions or removing functional dependencies.

For efficiency, the most efficient were also mostly the specialized passes and passes that focus on removing large elements. Examples for these are beta reducing expressions and removing declarations.

So the most valuable passes were those that were also early considered: removing declarations, turning types into the unti type and removing matches.

As an example for a bad pass, an older version of reducing type family applications was really bad, because it was applicable to all Haskell types in the program but only succeeded if there really was a type family application present, which is really rare. So it had a high number of test invocations, which made hsreduce spent a lot of time and a really low success rate, so it was rarely useful.

Pass name	t. i.	Δ identifiers	r.o.s.	efficiency
rmvDecls	9990	-11995	0.24704	-1.20070
type2Unit	70811	-7440	0.06359	-0.10506
rmvSigs	1811	-4801	0.57482	-2.65102
rmvMatches	7759	-4079	0.12694	-0.52571
simplifyType	10409	-2786	0.21817	-0.26765
expr2Undefined	48599	-2665	0.05520	-0.05483
type2WildCard	89891	-2293	0.02419	-0.02550
simplifyExpr	28649	-2201	0.06935	-0.07682
tyVarBndr	2607	-2022	0.72266	-0.77560
TypeFamilies.rmvEquations	1492	-1940	0.33914	-1.30026
TypeFamilies.apply	12512	-1440	0.05370	-0.11508
rmvConstructors	3166	-1430	0.26405	-0.45167
pat2Wildcard	11355	-1293	0.09660	-0.11387
filterExprSubList	3085	-1218	0.09983	-0.39481
rmvGuards	1237	-948	0.07518	-0.76637
contexts	2140	-824	0.14766	-0.38504
rmvFunDeps	311	-765	0.61093	-2.45980
simplifyConDecl	2656	-645	0.20745	-0.24284
TypeFamilies.rmvUnusedParams	1519	-633	0.19881	-0.41672
localBinds	2775	-626	0.09081	-0.22558
Functions.rmvUnusedParams	5256	-474	0.07534	-0.09018
TypeFamilies.familyResultSig	971	-473	0.36148	-0.48712
betaReduceExprs	239	-421	0.80753	-1.76150
unqualImport	1907	-416	0.11641	-0.21814
rmvTyClMethods	1073	-355	0.03355	-0.33084
rmvRHSs	9362	-238	0.00854	-0.02542
simplifyDerivingClause	540	-75	0.13888	-0.13888
etaReduceMatches	101	-64	0.38613	-0.63366
Typeclasses.rmvUnusedParams	517	-50	0.04448	-0.09671
rmvDerivingClause	421	-29	0.04038	-0.06888
rmvConArgs	2234	-14	0.00134	-0.00626
rmvImports	2030	-13	0.10738	-0.00640
handleMultiParams	3212	-12	0.00217	-0.00373
formatting	199	0	1.00000	0.00000
inlineType	61	0	0.44262	0.00000
splitSigs	976	8	0.00102	0.00819
Functions:inline	1687	1564	0.15826	0.92708
Arithmetic mean			0.21653	-0.41858
Median			0.11641	-0.21814

Table 5.3.: Comparison of the different passes;
abbreviations: total invocations (t. i.), rate of success (r.o.s.)

6. Conclusion and Future Work

We built hsreduce, a Haskell-specific test case reducer that beats domain-independent reducers like C-Reduce by up to 60 percent on single-file tests. This is due to it exploiting richer transformations like reducing type family applications, function inlining and turning values into stand-in dummy values. For isolating the effect of large dependencies, we also implemented a tool that merges imported modules into a single file, which significantly frees the user of the labor of having to do so manually.

There are still many things to explore that were not possible to realize due to time constraints. First, there is the merging of projects. Right now, hsreduce is able to merge some projects but there are bigger ones like GHC itself which are not possible to merge at this point.

Then there are still a lot of passes, which were out of scope for this work, one can implement. Examples are: dumping of single template haskell splices, normalizing types using the typechecker information and specializing polymorphic functions.

Additionally one could explore different search strategies more, for example to apply the same one that C-Reduce uses.

One could also write `Arbitrary` instances for the Haskell AST, to be able to fuzz Haskell programs, which can then be reduced by hsreduce. Another advantage would be to then be able to use QuickCheck's shrink function. If one would also write `Generic` instances for the GHC AST, then one could even use `genericShrink`. But to do this, there would be a lot of boilerplate code be involved and and it would be hard to balance terms, for example not to generate to deep App expressions.

Last, we feel it should be possible to build monadic pass combinators, similar to parser combinators. This might improve the usability of this tool and might unlock new synergies of combining existing passes.

Bibliography

[1] J. Regehr, Y. Chen, P. Cuoq, E. Eide, C. Ellison, and X. Yang, "Test-case reduction for c compiler bugs," in *Proceedings of the 33rd ACM SIGPLAN conference on Programming Language Design and Implementation*, pp. 335–346, 2012.

[2] A. Zeller and R. Hildebrandt, "Simplifying and isolating failure-inducing input," *IEEE Transactions on Software Engineering*, vol. 28, no. 2, pp. 183–200, 2002.

[3] D. MacIver, "structureshrink." `https://github.com/DRMacIver/structureshrink`, 2020.

[4] T. Ormandy, "halfempty." `https://github.com/googleprojectzero/halfempty`, 2020.

[5] R. Lämmel and S. P. Jones, "Scrap your boilerplate: a practical design pattern for generic programming," *ACM SIGPLAN Notices*, vol. 38, no. 3, pp. 26–37, 2003.

[6] G. Misherghi and Z. Su, "Hdd: hierarchical delta debugging," in *Proceedings of the 28th international conference on Software engineering*, pp. 142–151, 2006.

[7] R. Hodován, "picireny." `https://github.com/renatahodovan/picireny`, 2020.

[8] C. Sun, Y. Li, Q. Zhang, T. Gu, and Z. Su, "Perses: Syntax-guided program reduction," in *Proceedings of the 40th International Conference on Software Engineering*, pp. 361–371, 2018.

[9] "Cabal homepage." `https://www.haskell.org/cabal/"`. Accessed: 2020-12-13.

[10] "Stack homepage." `https://docs.haskellstack.org/en/stable/README/`. Accessed: 2020-12-13.

[11] "Gnu make homepage." `https://www.gnu.org/software/make/`. Accessed: 2020-12-13.

[12] "Shake homepage." `https://shakebuild.com/`. Accessed: 2020-12-13.

[13] J. Breitner, "hs-all-in-one." `https://github.com/nomeata/hs-all-in-one`, 2017.

[14] N. Broberg, “haskell-src-exts.” `https://github.com/haskell-suite/haskell-src-exts`, 2020.

[15] R. Cheplyaka, “hse-cpp.” `https://github.com/haskell-suite/hse-cpp`, 2020.

[16] “haskell-src-exts - no more releases.” `https://mail.haskell.org/pipermail/haskell-cafe/2019-May/131166.html`. Accessed: 2020-12-13.

[17] T. Tani, “Ghc renamer description.” `https://gitlab.haskell.org/ghc/ghc/-/wikis/commentary/compiler/renamer`, 2020.

[18] M. Pickering, N. Wu, and B. Németh, “Working with source plugins,” in *Proceedings of the 12th ACM SIGPLAN International Symposium on Haskell*, pp. 85–97, 2019.

[19] M. Pickering, “hie-bios.” `https://github.com/mpickering/hie-bios`, 2020.

[20] H. Organization, “Haskell language server.” `https://github.com/haskell/haskell-language-server`, 2020.

[21] H. Organization, “ghcide.” `https://github.com/haskell/ghcide`, 2020.

Erklärung

Hiermit erkläre ich, Daniel Krüger, dass ich die vorliegende Masterarbeit 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

Danke

Ich danke Sebastian Graf für die Unterstützung, Tipps und Anregungen.

A. Appendix

A.1. hsreduce Implementation

The implementation can be found here.

A.2. GHC Issues

Here are the issues corresponding to the test cases in our test case repository:

- T8763
- T13877
- T14270
- T14779
- T14827
- T15696
- T16979
- T18098
- T18140

Our test case repository sometimes has multiple versions of ticket.

Our test case for T8763 stems from this comment.

For T14827 it wasn't possible to get the original source code because it was in a now defunct bitbucket repository. Our test case is based on the self contained example posted there.

With T15696 we have three versions. T15696_1 stems from the first example posted there. T15696_2 contains a test case that resulted from using an earlier version of our merging utility. T15696_3 contains the containers repository as a submodule and can be used to test the merging utility. It should produce a file that is very similar to the one in T15696_2.

With T18140 we also have three versions. T18140_3 contains protocol-buffers-descriptor as a submodule and can be used to test the merging utility. T18140_2 contains a test case that resulted from using an earlier version of our merging utility. With that earlier version we weren't able to merge the whole project when we inlined the protocol-buffers dependency. That is why there is also a Text folder in there. C-Reduce can't work on that test case because it only copies the shell script and the test case to its temporary directories. T18140_1 resulted from taking the result of

applying *hsreduce* to T18140_2, putting that file as the main file of protocol-buffers-descriptor, inlining the dependency protocol-buffers dependency and merging (which now worked because the main file had now less imports.

All other test cases are based on the first examples posted on those tickets.