Karlsruher Institut für Technologie

Institut für Programmstrukturen
und Datenorganisation (IPD)

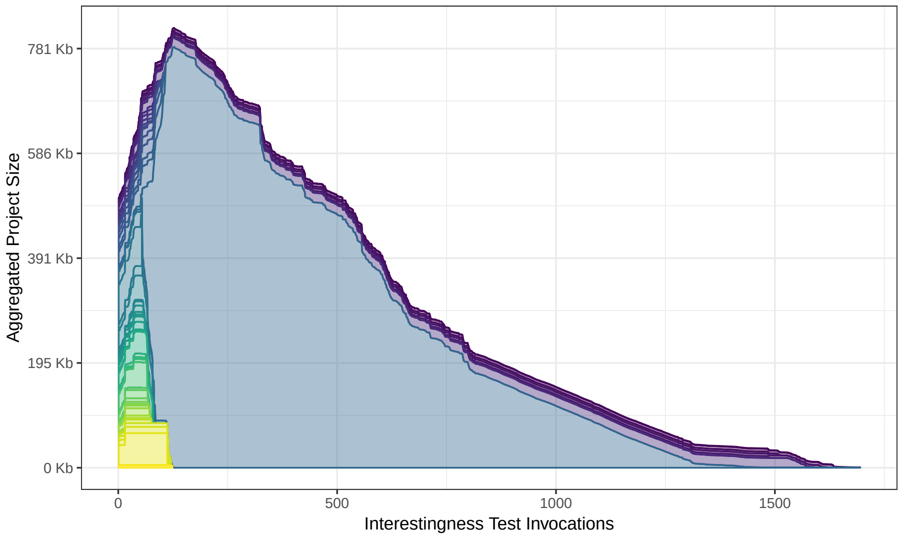
Lehrstuhl Prof. Dr.-Ing. Snelting

Elm-Reduce: Delta Debugging Functional Programs

Bachelorarbeit von

Philipp Krüger

an der Fakultät für Informatik



Erstgutachter:	Prof. Dr.-Ing. Gregor Snelting
Zweitgutachter:	Prof. Dr. rer. nat. Bernhard Beckert
Betreuende Mitarbeiter:	M.Sc. Sebastian Graf
Abgabedatum:	5. Juli 2019

Abstract

In many compiler issue trackers only bugs that have a minimal reproducible test case are investigated. Since the reduction of projects with thousands of lines of code to minimal test cases can be tedious to do manually, the barrier to reporting bugs is high.

Delta-debugging can be used to automate this process. We show that traditional delta-debugging tools are not sufficient to produce satisfyingly minimal reduction results, because it is necessary to not only remove parts of source code, but also replace them with language-specific constructs during reduction. The reduction speed can also be improved significantly by ensuring well-formedness, e.g. well-typedness, during reduction.

In this thesis, we develop the delta-debugging reducer Elm-reduce, which reduces Elm projects from megabyte sizes to test cases below 2 times the size of the optimal minimal test case. We also present reduction passes and techniques for implementing them in functional programming languages.

Zusammenfassung

In vielen Compiler Bug-Trackern werden nur Bugs bearbeitet, welche einen minimalen, reproduzierbaren Testfall haben. Da das Minimieren von tausendzeiligen Projekten zu einem minimalen Testfall allerdings sehr aufwändig sein kann, ist die Hürde zum Melden von Bugs hoch.

Delta-debugging kann dann verwendet werden, um diesen Prozess zu automatisieren. Wir zeigen, dass herkömmliche Delta-debugging Werkzeuge für die funktionale Programmiersprache Elm nicht ausreichend sind, um zufriedenstellend minimale Reduktionsergebnisse zu produzieren, weil es bei es der Reduktion notwendig ist, nicht nur Teile des Quelltexts zu löschen, sondern auch mit programmiersprachen-spezifischen Konstrukten zu ersetzen. Außerdem kann die Reduktion signifikant beschleunigt werden, wenn man bei jedem Reduktionsschritt ein gewisses Maß an Wohlgeformtheit, z.B. Typkorrektheit, erhält.

In dieser Arbeit entwickeln wir einen Delta-debugging Reduzierer Elm-reduce, welcher Elm Projekte von Megabytegrößen zu Testfällen, welche weniger als 2-mal so groß sind wie der optimale minimale Testfall, reduziert. Außerdem präsentieren wir Reduktionspässe und Methodiken, um diese in funktionalen Programmiersprachen zu implementieren.

Contents

1. Introduction

In August 2018 the 0.19 Version of the Elm compiler was released. Teams running Elm in production were eager to update because of great improvements in compiler speed. However, some users reported that it crashed when they tried to compile their projects that were of 35,000 lines of code or more[1].

From a compiler writer's perspective, this is an unfortunate situation: The bigger the projects crashing the compiler are, the harder it is to track down the root cause of the issue. Among all the source code she has to figure out what part actually caused it. Here, delta debugging comes into play: The whole source code is the test case and running the compiler is the test. Then, parts of the source code are deleted and the compiler is run again while making sure the error still occurs. Some have figured out this task manually and collected test cases that were as small as 5 lines of code.

Reducing source code while maintaining a compiler error is not an easy task: Consider this, not fully reduced program crashing the compiler when invoking it via `elm make File.elm --debug`:

```
1 import Array
2
3 type Message = Message (Array.Array ())
4
5 main : Platform.Program () () Message
6 main =
7     Debug.todo ""
```

Line based reduction would not be able to produce a smaller example in this case (except for blank lines). For example: Removing line 3 and therefore the `Message` datatype Definition would make this program ill-formed: The reference to `Message` in `main`'s type annotation couldn't be resolved. This causes the compiler to exit with a naming error - the correct behavior - before it even tries to generate code, which is where the fault lies.

It would even be impossible to produce a smaller example when reducing by character, except for renaming the 'Message' datatype or removing white space. Any other change to a character would make this program ill-formed, which would, again, prevent the bug form occuring. For this reason, current language-agnostic reducers that work only by removing characters perform very badly, producing test cases of over 50 times the size of minimal test cases (figure 4.1). It is especially hard for

[1] `https://github.com/elm/compiler/issues/1802`

strongly typed functional languages, as they have a particularly strong notion of well-formedness.

The key idea is to not only remove parts of the test case during reduction but to replace them with language-specific constructs. This can also significantly speed up reduction, as there is a higher chance of bugs triggering when the compiler does not exit early due to ill-formed source code.

Let us apply these key ideas to our example: We inline the type declaration for `Message`, substituting it with `Array.Array ()` everywhere it is referenced, obtaining a smaller test case that still crashes the compiler.

```
import Array

main : Platform.Program () () (Array.Array ())
main =
    Debug.todo ""
```

Of course, it is also possible to have bugs in the compiler code that checks the well-formed-ness. In these cases, it is still useful for reduction to keep well-formed-ness of the rest of the source code. It is not a requirement for Elm-reduce that the test case be parseable.

The goal of this thesis is to come up with an effective approach for reducing purely functional source code.

Our contributions are:

- A list of reduction passes, sufficient for most reductions in a simpler purely functional programming language, Elm.
- An approach to program reducer architecture in purely functional programming languages.
- Elm-reduce, an Elm project reducer, similar to C-Reduce, which reduces Elm compiler bugs to minimal test cases. It reduces projects to minimal test cases very similar to the test case above.

2. Basics and Related Work

2.1. The Elm Programming Language

Elm-Reduce is built specifically for reducing projects in the Elm Programming Language. Elm started as Evan Czaplicki's Senior thesis [1] but has since grown to be a language used for commercial web-development[1].

Conceptually, Elm is part of the classic 'ML'-family of programming languages, featuring purity, lambdas, and higher-order functions. Function definitions look very similar to Haskell's:

```
member : a -> List a -> Bool
member x xs =
    List.any (\a -> a == x) xs
```

Figure 2.1.: An Elm function that checks whether a given element exists inside of a given list. Function arguments are on the left-hand side of the equation and function application is denoted by a space after `List.any`. Functions are curried. The `(\arg -> body)` syntax is used to construct anonymous functions (lambdas).

Elm has many properties common for ML-family languages, but uncommon for languages, for which reducers are currently used or have been developed for, like C, Javascript, Java or similar:

Purely Functional: It is impossible to change the value of bound names. (Global) State must be handled explicitly by passing it to functions using it. All user-defined functions must be side-effect free. This means that Elm is referentially transparent.

Static Typing: Every value's type is and must be known at compile time (including polymorphic types). If there is any function call which is ill-typed, elm will generate a type-error at compile time before generating code. Elm has a Hindley-Milner type system with limited extensible record typing (a subset of Daan Leijen's 'Extensible Records with Scoped Labels' [2]), which allows it to infer types for any valid term. In this thesis, we reduce a test case with a bug in the implementation of this type inferencer.

[1] `https://elm-lang.org`

Record Types: Elm has some structural typing via its record types. These types don't have to be declared but can be used in-place like the function arrow or tuple type constructor. However, they are most commonly used inside type-aliases:

```
type alias Person =
    { name : String
    , age : Int
    }
```

Algebraic Datatypes (ADTs): One can also define nominal sum-of-product types via a `type` declaration (called `data` in Haskell and `datatype` in Standard ML):

```
type Result error value
    = Ok value
    | Err error
```

Compiled/Transpiled to Javascript: This makes it possible to run in the browser. Once the Elm compiler succeeded compiling an application, a single Javascript file is generated. Elm doesn't use pre-built library artifacts when compiling applications with libraries but builds a whole project from source. This means that applications need access to their dependencies' source code.

2.1.1. Terminology

Throughout this thesis, we refer to syntactic constructs and other terminology of various kinds. We explain used terminology to avoid confusion:

Expression The right-hand side of the function definition in figure 2.1 is its body, which we call expression. These are also often referred to as terms.

Subexpression An expression's subexpression is one of the expressions it consists of. The body of a lambda, a case of a case-of expression, the function or argument in an application or the right-hand side of a field in a record constructor are subexpressions.

Type Expression The right-hand side of the type-of operator (:) in figure 2.1 is a type expression. Type expressions can also occur on the right-hand sides of type aliases or in fields of type constructors.

Type Subexpression Just like expressions, type expressions can be nested by type application. A type application's arguments are what we refer to type subexpressions.

Module Dependency Graph A graph in which nodes are modules and directed edges the dependencies going from referrer to dependency. Only acyclic module dependencies are allowed by the Elm compiler.

Function Call Graph A graph in which nodes are functions and edges are possible calls from a caller to the called function.

Declaration Dependency Graph A graph in which nodes are either functions or type declarations. Edges denote references from referrer to target. Functions are always sources in this graph, as only type declarations can be referred.

2.1.2. The compiler architecture

The elm compiler is a multi-pass compiler. Its phases are roughly the following:

- Parsing directly into a 'Source' AST
- Validating a Source AST to be a 'Valid' AST: Validation ensures that there are no stale type annotations, that ports (for foreign Javascript interfacing) are only used when the module is declared as 'port module' and similar.
- Resolving module imports and naming to a 'canonicalized' Module: This annotates each identifier with the module and package in which it was defined.
- Type checking: Function definitions are type-checked against their declared type annotation, and if none is given, their type annotation is inferred. When any definition is ill-typed, an error is raised.
- Optimization: An 'Optimized' AST is produced, which only includes code that can be reached in the call graph from the 'main' definition and includes some transformation passes preparing for compilation, for example, tail-call optimization.
- Code generation: All code is rendered as javascript source code. The compiler reuses cached Optimized ASTs from previous runs of the compiler, skipping the compilation procedure for modules that didn't change.

The compiler executes these phases for each module individually. The modules get compiled in topological order: Every module's dependency gets compiled before itself.

This means that given a module A that depends on B, module B is fully compiled before module A even begins type-checking or canonicalization phase.

2.2. Delta Debugging

Delta debugging, first developed by Andreas Zeller [3], is a methodology to narrow down on failure-inducing input for a test case. When its input can be described as a set of changes, or – for our purposes – tokens, the input can be shrunk by removing tokens and asserting that the test case still fails.

Known examples of delta debugging are the `git bisect` tool, in which commits are changes that can be removed, the QuickCheck `shrink` function for shrinking random test cases or various compiler bug reduction tools that try to remove syntax tokens from a test case while still crashing the compiler.

As delta debugging can be used for a variety of purposes, test cases are not referred to as ‘successful’ or ‘failing’, but ‘interesting’ or ‘uninteresting’. The reduced test cases are then usually checked for interestingness by an executable script that exits with 0, when the test is interesting, or with a non-zero exit code otherwise. Delta debugging algorithms can also be used to isolate failure-inducing input by reducing an interesting and uninteresting test case side-by-side, while reducing the difference between these, as is the case for `git bisect`, for example. Then, interestingness tests can also report a test case as ‘unresolved’. In this thesis, we are only concerned with producing minimal failing test cases, therefore our interestingness tests don’t report unresolved-ness.

A test case is considered ‘1-minimal’ when it is impossible to remove any token or change from it while keeping the test case interesting. We will use the term ‘minimal’ to refer to ‘1-minimal’ in the rest of this thesis. We will use ‘optimal minimal’ test case when it is the smallest test case we know of that reproduces an issue. Any optimal minimal test case is therefore also 1-minimal.

In 2002, new delta debugging algorithms with very good asymptotic performance (in terms of interestingness test invocations) were developed [4] and in 2006, ‘Hierarchical Delta Debugging’ [5] was developed for performing better on tree-like test inputs.

2.3. C-Reduce

C-Reduce [6] is a test case reducer for C source code. Historically, C source code was reduced using the input-format agnostic suite of reducers developed in [4]. However, those would sometimes produce C source code with undefined behavior[2]. To avoid undefined behavior during reduction, C-Reduce was developed. After every reduction, C-Reduce asserts the defined-ness of the reduced C source code. Additionally, many C-specific reduction passes were developed: Inlining functions, deleting function bodies, lifting variables from function to global scope or removing a function parameter from its definition and all call sites, for example. C-Reduce also attempts the original DDmin reducers in-between those passes.

Their evaluation [6] shows that they not only have guaranteed defined behavior for their minimal test cases but also substantially improved reduction speed and quality.

C-Reduce can be invoked with the `--not-c` flag, which turns off all C-specific reduction passes, allowing us to run C-Reduce on Elm source code and with it all DDmin algorithms as well.

[2]See the C99 language standard `http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1256.pdf`, section J.2 ‘Undefined Behavior’.

2.4. Fixed Point

Elm-reduce's main looping mechanism works similar to C-Reduce: It repeatedly tries to apply reductions, until no more reductions can be applied or reduced projects become bigger than the original. This can be mathematically described by a fixed point. Here we introduce relevant definitions and theorems.

Definition 1. *Given a function $f : A \to A$, a fixpoint (fixed point) $a \in A$ is defined by a solution to the fixpoint equation:*

$$f(a) = a \tag{2.1}$$

Definition 2. *A Complete Lattice is a partially ordered set L, with the following property: Each subset $M \subseteq L$ has a least upper bound $sup\ M$ and a greatest lower bound $inf\ M$.*

Theorem 1. *Knaster-Tarski Theorem: Given a complete lattice L and a function $f : L \to L$ with the monotony property $\forall x, y : x \leq y \Rightarrow f(x) \leq f(y)$, then the fixed points of f form a complete lattice.*

From this theorem follows in particular that there is a greatest and a smallest fixpoint.

2.5. The Applicative Functor

In section 3.5 we develop an applicative functor aiding us in writing reductions, hence we introduce it here.

Applicative functors were first described in a functional pearl by Conor McBride and Ross Paterson [7]. Applicative functors are more powerful than functors but less powerful than monads. Every monad is also an applicative functor.

Applicative functors require two operators, one operation that – intuitionally – lifts a value to the applicative context, and an operation that performs function application with a function and an argument in a context.

In Haskell applicative functors are defined in the standard library as a type class (see figure 2.2), where the lifting function is defined as `pure` and the application operation is implemented as an infix operator `<*>` so that using this syntax is similar to ordinary function application with infix spaces.

The applicative functor has to obey the following laws:

$$\begin{aligned} pure\ id <\!*\!> v &= v, && \text{(Identity)}\\ pure\ (\circ) <\!*\!> u <\!*\!> v <\!*\!> w &= u <\!*\!> (v <\!*\!> w), && \text{(Composition)}\\ pure\ f <\!*\!> pure\ x &= pure\ (f\ x), && \text{(Homomorphism)}\\ u <\!*\!> pure\ y &= pure\ (\$\ y) <\!*\!> u, && \text{(Interchange)}\\ \forall u, v, w. \end{aligned}$$

(where $(\circ)$ is function composition and $(\$\ y)$ is a function that applies y to its argument)

```haskell
class Functor f => Applicative f where
    pure :: a -> f a
    (<*>) :: f (a -> b) -> f a -> f b
```

Figure 2.2.: The (simplified) type class definition of an applicative functor in the Haskell standard library.

3. Architecture and Implementation

At the time of writing, all source code and installation instructions of Elm-reduce can be found at `https://gitlab.com/matheus23/elm-reduce`.

Before diving into an implementation overview or architecture, it makes sense to present Elm-reduce in terms of its user interface, so one can better imagine how all gears line up inside of the Implementation.

3.1. User Interface

Elm-reduce, from the user interface perspective, is a single executable `elm-reduce`, which can be executed in the directory containing the Elm project of interest. It takes the path to a shell script as a parameter. This shell script is the so-called 'interestingness' test. This is an example for one of the shell-scripts we run:

```
#!/bin/sh
rm -r elm-stuff
EXPECTED_ERR=
    "elm: Map.!: given key is not an element in the map"
elm make --output=elm.js --debug src/Main.elm 2>&1 |
    grep -qF "$EXPECTED_ERR"
```

This shell script first cleans the `elm-stuff` directory used by the elm compiler as compilation cache, then runs the elm compiler using the `--debug` flag, and finally checks whether the string *„elm: Map.!: given key is not an element in the map"* occurred in the compiler's output. This is the error message generated by the elm compiler on a particular compiler bug. Crucially, this script exits with exit code 0, when the error occurred, marking the current project state as 'interesting', and exits with exit code 1 otherwise. This is accomplished by `grep`s behavior when passed the 'quiet' flag (`-q`).

Once started, the executable reports statistics about the current reduction progress:

```
[...]
Variant: Removed constructor KintoError
Reduced: 56.33% (10665 bytes)
Variant: Removed constructor field
Reduced: 56.22% (10644 bytes)
Variant: Deleted type declaration Msg
Detected errors in 1 module.
[...]
```

```
Could not reduce code size further.
Reduced code size: 1182 bytes(first invocation: 18932 bytes)
```

Elm-reduce will create a git commit for every reduction that resulted in a project state that was still considered interesting. Having a git project initialized in the project directory is therefore a prerequisite. Also, one doesn't have to worry about Elm-reduce changing the current project irreversibly, since it will only produce new commits and never alter the previous git history.

As soon as there are no more reductions available, or all reductions result in an uninteresting test case, Elm-reduce finishes and reports reduction statistics. The resulting test case is then 1-minimal.

3.2. Integration with existing Solutions

Elm-reduce is written in Haskell, since the Elm compiler and the elm-format utility are written in Haskell too. Because we're doing abstract syntax tree (AST) transformations, but want to test for compiler errors (or any other interesting test cases) via a simple shell script, we need to both have the elm project available as AST and as source file on disk. We use the Elm compiler's parser for parsing the file and elm-format's pretty printing for printing the file back to disk.

Initially, implementing Elm-reduce in terms of additional passes for C-Reduce was considered, but ultimately decided against, because our high-level passes for reducing modules by merging them would not fit the single-file architecture of C-Reduce.

3.3. Reduction loop Architecture

Before Elm-reduce begins its main loop that tries to find a fixpoint of reduction, it prepares the Elm Project by vendoring dependencies and formatting all source files. It also has passes like merging or deleting Elm modules that work across the whole project and passes that work on single files. This results in two nested loops of iteration as shown in figure 3.1.

Elm-reduce has a number of different reduction passes, each doing a certain kind of code transformation. Removing dead definitions, reducing case-of expressions or removing union type constructors for example. Some of these reductions can enable more reductions for a different reduction pass, therefore it would not suffice to just run each pass once. It is not easy to figure out which new reductions will be enabled, because that is highly dependent on the bug tested, and therefore the interestingness test.

For these reasons Elm-reduce works similarly to C-reduce: It will try a list of all different passes after each other repeatedly. Each time making sure the resulting project is shrunk in terms of some metric. The metric chosen for Elm-reduce is bytes for all elm modules and the project's `elm.json` file.

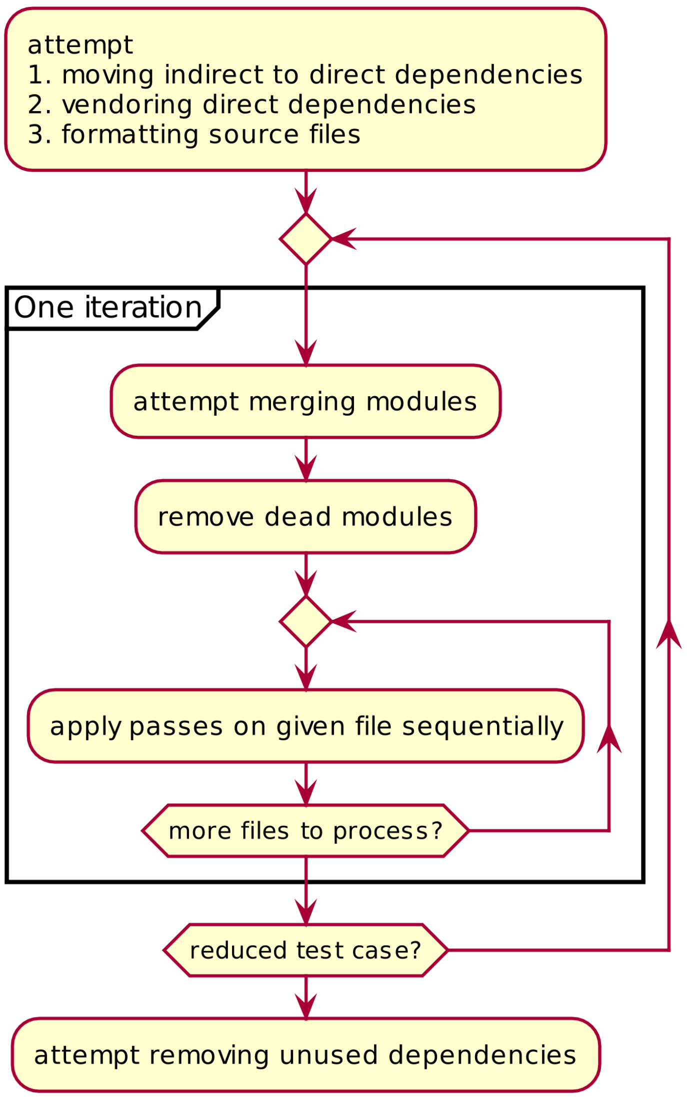

attempt
1. moving indirect to direct dependencies
2. vendoring direct dependencies
3. formatting source files
One iteration
attempt merging modules
remove dead modules
apply passes on given file sequentially
more files to process?
reduced test case?
attempt removing unused dependencies

Figure 3.1.: Elm-reduce's main loop architecture. 'One iteration' is what we modeled as the function f that we compute its fixed-point of. Note that after any pass we check the interestingness test and either `git commit` or `git reset` changes.

Because of the Knaster-Tarski Fixpoint Theorem, this loop is guaranteed to terminate, as long as all passes take a finite amount of time:

- We model L as the set of all possible elm projects, which becomes a complete lattice, when equipped with a metric $m(p) = \texttt{byteSize}(p)$.
- Let $r : L \to L$ be the function applying all reduction passes to L one after another once.
- We model $f : L \to L$ as the function applying all reductions via r, as long as this would reduce the metric m, or behave like the identity function otherwise:

$$f(p) = \begin{cases} r(p), & \text{for } m(r(p)) < m(p) \\ p, & \text{for } m(r(p)) \geq m(p) \end{cases}$$

It also follows that the iterated loop computes the greatest fixpoint p_{fix}. Of course, one would hope to find the smallest fixpoint, resulting in a smaller reduced project. We don't know of a reliable way to compute the smallest fixpoint, however.

Therefore, we suggest a different way of developing reducers to achieve better results: Assume, there is a smaller fixpoint p_{small}, but Elm-reduce only reduced to the fixpoint p_{fix}. The idea is to change f to f' in a way, so that $f'(p_{fix}) < p_{fix}$, and that $f'(p_{small}) = p_{small}$. So instead of trying to find another way to compute fixpoints, we would try to find better reduction functions, resulting in fewer possible fixpoints, therefore hoping that the greatest fixpoint is also the only fixpoint.

It is possible that a fixpoint p_{small} wouldn't be reachable from f' anymore, since there needed to be another choice made earlier in f's history, but in our experience, looking at p_{fix} and guessing p_{small} made many reduction passes obvious and improved Elm-reduce's reduction behavior.

3.4. Passes

We categorize reduction passes similar to previous work [8] as either destructive or non-destructive: They either preserve program semantics or don't. These two kinds of passes work in synergy: By destroying program semantics, destructive passes break up dependencies between function declarations enabling reductions from non-destructive passes.

3.4.1. Non-destructive Passes

These passes don't change program semantics: If they were applied to a project and run, it should run identically to before. Some of these passes are the main mechanism through which bytes are removed from projects, e.g. the dead module or dead definition removal passes, and their applicability is mostly enabled by destructive passes.

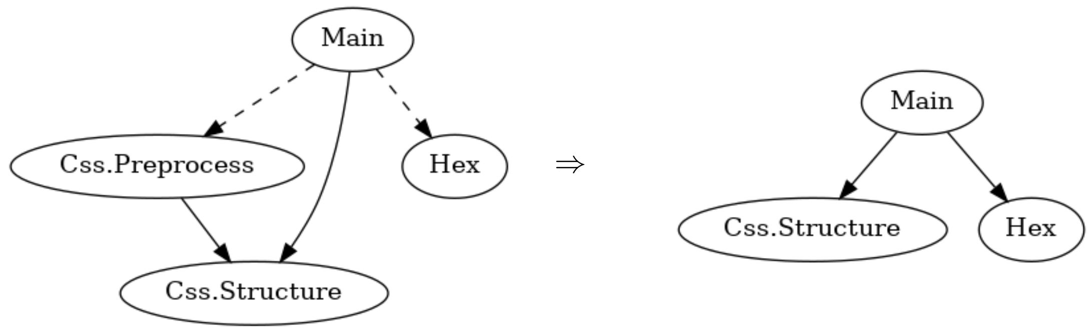


Figure 3.2.: The module dependency graph of a test case project mid-reduction before (left) and after (right) merging the module `Css.Preprocess` into the `Main` module. Dotted edges indicate reducible points. Alternatively, the `Hex` module could be merged, but the `Css.Structure` module couldn't, because it has more than one referrer.

Note that any reduction on types in Elm is categorized as non-destructive by this definition, because type declarations does not change execution, except for rare cases when using Elm's `comparable`, `appendable` or `number` type variable special cases.

Formatting

The very first pass Elm-reduce attempts is formatting all files using elm-format[1]. Since Elm-reduce uses elm-format as a library for printing out reduced variants of Elm code internally, it is a prerequisite that the bug is not triggered by anything related formatting. This pass ensures this baseline.

Merging Modules

This pass analyzes the module dependency graph and copies all function definitions, type declarations, imports and exports from a module into its only referrer, if that is the case. This enables passes, which would be too complicated across module boundaries to work on the combined module. Ideally, and usually, as one of the first passes, Elm-reduce combines all of a project's modules into a single file.

Figure 3.2 shows how the module dependency graph is simplified by merging a module into the main module. Also notice that doing a single merging step opens up new possibilities for merging further modules. Given all merges are successful, it is always possible to merge all modules of a project into a single module using these incremental steps (see appendix A.1 for proofs).

[1]`elm-format` is the quasi status-quo formatter for Elm source code: `https://github.com/avh4/elm-format`

Deleting Dead Modules

When modules could not be merged for some reason, but there is no more dependency on a module, this pass deletes it.

Vendoring Dependencies

This reduction copies a dependency's source code into the project and links to it directly, instead of referring to it in the project's `elm.json` file. This process has been referred to as dependency vendoring, but could also be described as 'inlining' dependencies, similar to how functions can be inlined.

This enables the reduction of code triggering bugs inside of dependencies. It is also noteworthy that the "Map.!" bug will vanish once a specific dependency has been vendored.

Removing Dead Function Definitions

Usually, about 80%[2] of Elm code consists of function definitions. Removing dead function calls is the biggest factor in code reduction. This pass analyzes the call tree inside of a module to determine functions which are never referenced by any other function definition.

Removing Dead Type Declarations

This pass works analogous to removing dead functions: Type declarations are removed, when no function type annotation or other type declaration references it anymore. Both type aliases and union type declarations are removed in this pass.

Reducing Union Type Declarations

This pass reduces type declarations by removing constructors, removing fields from constructors or reducing types inside the fields of constructors. Only constructors that are not matched or constructed and therefore dead can be successfully removed, because the interestingness test usually causes the Elm compiler to test for well-typed-ness.

Figure 3.3 shows what multiple such reductions applied on an example would look like.

Converting Union Types to Type Aliases

When a type declaration is reduced so that it only has a single constructor with a single field, this pass transforms it to a type alias with that single field's type for its body. This enables further reduction that is only defined or possible for type aliases.

[2]This number was determined as the ratio of function definition lines to total lines of code on one of the open source test projects "webvim": `https://github.com/vim-dist/webvim`, commit `b2f103c8`

```
type Response
    = Failure String                 type Response
    | Success Int Payload      ⇒         = Success Payload
    | Info (Maybe Int)                   | Info Int
```

Figure 3.3.: Examples of union type declaration reductions: Removing the `Failure` constructor, the `Success` constructors first field and reducing the `Info` constructors first field.

```
type Response                  ⇒     type alias Response =
    = Success Payload                    Payload
```

Reducing Type Aliases

In Elm, most type alias definitions are aliases for record types. This pass tries to remove record type fields or reduce the fields' type expressions themselves by replacing expressions with one of their subexpressions.

```
type alias Payload =
    { route : String                 type alias Payload =
    , page : Maybe Int         ⇒         { page : Int
    }                                    }
```

Inlining Type Aliases

This pass substitutes all usages of type aliases with their body. This can eliminate the type-alias being the source of a bug.

```
type alias Model =
    { name : String                  main
    }                          ⇒         : Program () Msg
                                             { name : String }
main : Program () Msg Model
```

Reducing Type Expressions

In this pass type expressions in function type annotations, union constructor fields and type aliases are replaced by one of their subexpressions.

```
tail                                 tail
    : List a                   ⇒         : List a
    -> Maybe (List a)                    -> Maybe Int
```

Removing Imports

This pass removes unused imports from a module. This is both for aesthetic reasons, since every other pass is implemented using the Elm compilers parser, which automatically adds default imports to parsed modules, and for enabling further optimization by deleting now unreferenced modules.

Removing Dead Dependencies

This pass removes dependencies from the `elm.json` file, for both aesthetic reasons and for eliminating dependencies as the cause of issues.

Moving Indirect to Direct Dependencies

This pass moves dependencies in the `elm.json` file from the 'indirect' section to the 'direct' dependency section. The indirect dependency section lists all dependencies-of-dependencies of a project. They are not exposed to the current project and cannot be imported, but can be imported by its dependencies. Moving dependencies from the 'indirect' section to the 'direct' section causes the "Map.!" to vanish.

3.4.2. Destructive Passes

These three passes change program semantics to enable further optimizations. While they themselves don't contribute much to the reduction directly, they are crucial. If these passes would not be applied, the program semantics would have to stay the same throughout reduction, and its impossible to reduce a project substantially without changing semantics.

Stubbing

This pass replaces function definition bodies with a stub. This stub is an expression which type-checks against any type in elm, its essentially an infinite loop:

```
let
    undefinedValue_ _ =
        undefinedValue_ ()
in undefinedValue_ ()
```

This expression uses an unusual identifier to avoid shadowing any top-level definitions, as that is not allowed in Elm.

Removing function bodies is the best way to simplify the function call graph, as this removes all outgoing edges from a node.

Reducing Case Expressions

This pass simplifies the patterns matched in case expressions, removes cases from case expressions or replaces a case expression by one of its cases' body completely.

```
update msg m =
    case msg of
        PaymentAdded p ->
            ( emptyPayments
                msg m
            , Cmd.none
            )

        Refetch ->
            ( m
            , fetchPayments
                Fetched
                KintoError
                m.client
            )
```

⇒

```
update msg m =
    case msg of
        _ ->
            ( emptyPayments
                msg m
            , Cmd.none
            )
```

Figure 3.4.: Example of two steps of the case reduction pass. The `PaymentAdded` constructor match was replaced with a wildcard. As a result, the `Refetch` case could be removed. Note that this code is a simplified variant of some intermediate step in the reduction of a test case project, hence doesn't make much sense anymore.

This can reduce dependencies on some constructors or types of their fields, enabling more reductions with union type or type alias reduction.

Figure 3.4 shows an example of this pass.

Reduce Expressions to Subexpressions

This pass works similar to the type expression reduction pass, but on value-level expressions. It replaces expressions with one of their subexpressions. For example, an if-expression is replaced with either its then or else branch, a let definition with its body or one of its definitions or a function call with one of its arguments.

This reduction is attempted at any depth level of an expression, therefore there are many ways to apply it, making it infeasible to apply at early stages, but instead it is applied as one of the last passes, when the code base already shrunk substantially.

The following example is adapted from a real-world reduction case. Notice, how this pass can change types of expressions, for example, changing the inferred type of the `update` function. In this case, this still triggered a bug in the Elm inferencer, therefore still keeping the test case interesting.

```
update msg m =
    ( emptyPayments msg m          ⇒   update msg m =
    , Cmd.none                              emptyPayments msg m
    )
```

3.5. Reduction combinators

Reducing abstract data types for ASTs was a common task while implementing the goals for this thesis in elm-reduce. Often we were confronted with situations, in which we wanted to reduce small parts of our ASTs, while leaving irrelevant information intact. In this thesis we used an applicative functor we call `Reductions`, which we later realized was first defined by Joachim Breitner[3]. Writing reductions in applicative style and interpreting reductions as an effect greatly reduces code complexity. For this reason we present an introduction to them in this thesis.

Lets say we want to reduce the list of fields of an elm constructor. Working only on a list of elements, we can produce a list of variations like following:

```
reduce :: [a] -> [[a]]
reduce [] = []
reduce (x:xs) = xs : map ((:) x) (reduce xs)
```

Another important requirement was that we would only do one reduction at a time. If we were doing more than that, we might not know which reduction caused our test to become uninteresting. Note that `reduce` only removes one element at a time at maximum:

```
> reduce [1,2,3]
[[2,3],[1,3],[1,2]]
```

However, if we want to produce reductions of a pair of lists, one would think of the list monad as a way to combine reductions of first elements and reductions of the second elements, but this would always generate variants with multiple reductions applied at once:

```
> traverse reduce (Pair [1,2,3] [4,5,6])
[Pair [2,3] [5,6],Pair [2,3] [4,6],...]
```

Instead, we want to accumulate different variants, reducing either side of a pair. We can do this by reducing its parts and accumulating the results, mapping the original values to them:

```
>    map (flip Pair [4,5,6]) (reduce [1,2,3])
> ++ map (     Pair [1,2,3]) (reduce [4,5,6])
[Pair [2,3] [4,5,6],...,Pair [1,2,3] [5,6],...]
```

[3]Published as the library „successors” on hackage: `http://hackage.haskell.org/package/successors`.

```haskell
data Union = Union String [String] [(String, [Type])]

unionReductions :: Union -> [Union]
unionReductions (Union name typeParams constructors) =
    map (Union name typeParams)
        (reduce constructors
        ++ reduceElems constrReductions constructors)

reduceElems :: (a -> [a]) -> [a] -> [[a]]
reduceElems reduceElem [] = []
reduceElems reduceElem (x:xs) =
    map (\x' -> x':xs) (reduceElem x)
    ++ map (\xs' -> x:xs') (reduceElems reduceElem xs)

constrReductions :: (String, [Type]) -> [(String,[Type])]
constrReductions (name, fields) =
    map (name,) (reduce fields)
```

Figure 3.5.: Defining the list of reductions of a Union type without the 'Reductions' Applicative. It is crucial that only one reduction happened in the output list of variants (which is why the list applicative is not an option). Note the similarity of `reduceElems`' signature to `traverse`'s.

This works, but is significantly more cumbersome than simply traversing with reduce. The clumsiness becomes very apparent when we try to do this with some semi-real-world code, like reducing union constructors and their fields (see Figure 3.5).

Let us instead define a reductions datatype with an `Applicative` and `Semigroup` instance and a function that generates reductions by removing elements of a list.

```
data Reductions a = Reductions
    { original :: a
    , variants :: [a] }
    deriving (Show, Functor)

instance Applicative Reductions where
    pure x = Reductions x []
    (Reductions f f') <*> (Reductions a a') =
        Reductions (f a)
            (map ($ a) f' ++ map f a')

instance Semigroup (Reductions a) where
    (Reductions a a') <> (Reductions _ b') =
        Reductions a (a' <> b')

listReductions :: [a] -> Reductions [a]
listReductions [] = Reductions [] []
listReductions (x:xs) =
    Reductions (x:xs) [xs]
    <> fmap ((:) x) (listReductions xs)
```

`traverse` now only applies one change at a time:

```
> variants $ traverse listReductions (Pair [1,2,3] [4,5,6])
[Pair [2,3] [4,5,6]
,Pair [1,3] [4,5,6]
,Pair [1,2] [4,5,6]
,Pair [1,2,3] [5,6]
,Pair [1,2,3] [4,6]
,Pair [1,2,3] [4,5]]
```

The benefits of having an `Applicative` are not only being able to use `traverse` but also that they integrate well with lenses and prisms! See figure 3.6 for a real-world side-by-side comparison.

In appendix A.2 we prove the `Reductions` applicative functor to be lawful.

```haskell
unionReductions' :: Union -> Reductions Union
unionReductions' (Union name typeParams constructors) =
    Union name typeParams <$>
        (listReductions <> traverse (_2 listReductions))
        constructors

-- more general variant also defined in 'lens' package
_2 :: Functor f => (a -> f b) -> (x, a) -> f (x, b)
_2 f (x, a) = fmap (x,) (f a)
```

Figure 3.6.: Writing a reducer for union type constructors using our `Reductions` Applicative. Proofs for its instances can be found in appendix A.2 and A.3.

4. Evaluation

In this chapter, we evaluate Elm-reduce in terms of reduction quality and reduction speed. We begin by introducing the different test cases in terms of bugs and projects, try to evaluate existing solutions like C-Reduce and finally look at our results.

Even though we try to compare Elm-reduce to C-Reduce, no reducer can handle whole Elm projects by itself, except for Elm-reduce, which makes comparisons tricky.

4.1. Test Cases

In this section, we will discuss the test cases we chose to evaluate. Test cases are composed of a compiler bug and a project that triggers that bug. Some of our test cases are 'real-world', in the sense that these test cases have occurred for users during development without them intending to trigger the bug. Others were test cases we manufactured by implanting compiler-bug-triggering code into real-world projects.

4.1.1. Elm Compiler Bugs

It is important to test Elm-reduce with as many bugs as possible because different bugs demand different reduction behavior. For example, some bugs happen during compilation and some during the type-checking phase. Elm-reduce won't successfully trigger a code-generation bug when generating an ill-typed variant, but can successfully trigger a bug with an ill-typed variant when the bug occurs during type-checking. Of course, Elm-reduce doesn't have to know about which phase the bug is triggered in or whether the code base is currently well-typed, it merely knows whether the bug occurs or not, but that will influence which passes are successful.

For every compiler bug, we will present a minimal test case that is still human-readable and formatted with `elm-format`, so that identifier length does not affect minimality, and clever formatting only has a limited impact.

'Map.!' crash

This bug is widely known in the Elm community. There are multiple bug reports for it in the Elm compiler issue tracker, this meta-issue lists some: `https://github.com/elm/compiler/issues/1851` One of many minimal test cases for this bug is following code:

```elm
module Main exposing (main)

import Array


main : Platform.Program () () (Array.Array ())
main =
    Debug.todo "..."
```

This bug can be triggered by referencing an opaque, higher-kinded type from an imported library, in this case, `Array` from `elm/core`. The compiler has to be invoked with the `--debug` flag, and must be brought into code-generation phase (the `--output` parameter must not be `/dev/null`) so that the bug occurs:

```
$ elm make src/Main.elm --output=elm.js --debug
Success! Compiled 1 module.
elm: Map.!: given key is not an element in the map
CallStack (from HasCallStack):
    error, called at ./Data/Map/Internal.hs:610:17 in
    containers-0.5.11.0-[hash]:Data.Map.Internal
```

This bug is interesting to us, because it affected lots of users and happens during code generation, requiring well-typed code in all reduction steps. Program semantics – and therefore function bodies – are completely irrelevant to this bug, which makes it a special case for our reducer.

'index-out-of-bounds' crash

This bug was reported in the compiler issue `https://github.com/elm/compiler/issues/1890`. A slightly smaller test case than the reported one is the following:

```
1  module Main exposing (emptyPayments, update)
2
3
4  emptyPayments model =
5      let
6          extendedIdx =
7              model.expandedIndex
8      in
9      update model
10
11
12 update model =
13     let
14         paymentsBefore =
15             model.payments
16     in
17     emptyPayments
```

This seems to trigger a bug during type inference with mutually recursive functions and extensible records. The compiler crashes before it can type-check all expressions, so its minimal test case does not have to be type-correct (notice the missing argument to emptyPayments in line 14).

```
$ elm make Main.elm
elm: ./Data/Vector/Generic/Mutable.hs:703 (modify):
index out of bounds (3,3)
CallStack (from HasCallStack):
    error, called at ./Data/Vector/Internal/Check.hs:87:5 in
    vector-0.12.0.1-[hash]:Data.Vector.Internal.Check
elm: thread blocked indefinitely in an MVar operation
```

We chose to investigate this bug because it has quite a complex minimal test case with multiple top-level definitions.

‘out-of-memory’ crash

This bug was first reported in an Elm compiler issue: `https://github.com/elm/compiler/issues/1700`. A minimal test case is the following:

```
module Main exposing (listUnique)


listUnique elem unique =
    if List.member elem unique then
        unique

    else
        unique :: elem
```

This program would be well-typed if the variable names 'unique' and 'elem' were switched. If not switched, they cause this expression to have an infinite type. During type-checking, elm gets stuck in an infinite loop and consumes unlimited amounts of memory:

```
$ ulimit -v 2048000 # Limit virtual memory usage
$ elm make src/Main.elm
elm: out of memory
```

We chose to investigate this bug because it has to be reduced at the expression level.

4.1.2. Elm Projects

It is very hard to find whole projects to reduce that trigger compiler bugs. In the Elm compiler issue tracker, many compiler bugs are reported, but these only contain the minimal, reduced test cases, usually a single file size between 10-20 lines of code. Some issues mentioned, they had to manually reduce their project of sizes above 30 thousand lines of code[1], but it is hard to gain access to these because they are proprietary code.

We looked through commit messages on GitHub mentioning 'Map.!' and 'Elm' and found some projects to test, as well as one project of our own that triggered the same compiler bug.

we-connect: `https://github.com/dillonkearns/we-connect`, modified not to vendor a library so that this project still triggers the bug. 2510 lines of code[2].

webvim: `https://github.com/chendesheng/webvim`. 15513 lines of code. A web-based code editor with vim inputs.

flatawesome: `https://gitlab.com/matheus23/flatawesome-elm`. 2006 lines of code. A single-page-app for managing a shared flat.

[1] `https://github.com/elm/compiler/issues/1802`

[2] Measured using the command-line utility 'cloc', does not include empty lines or comments.

4.2. Results

When evaluating reduction on Elm projects we use the count of interestingness test (test script) invocations (named ‘steps’ in the following table) opposed to CPU time as metric for performance. This has multiple advantages:

- It is more independent of system resources than CPU time.
- We can compare reductions on the same code base for different test scripts and obtain results independent of the test script execution times. We can use this to compare reductions for different bugs.
- The same metric is used by other delta debugging research [4].

We also use lines of formatted code as a metric for the quality of reduction, because that makes the metric independent of identifier naming and formatting choices. The formatting style is determined by the non-configurable `elm-format` formatter with only a limited way of influencing newline placement.

4.2.1. Naive Approach

At first, we take a look at a naive approach using C-Reduce to do most of the reduction. We chose to compare to C-Reduce because its `--not-c` option makes it a great general purpose reducer and there exists no other Elm-specific reducer today. Unfortunately, C-Reduce cannot be run on multiple input files easily[3]. Usually, C-Reduce is run on C projects by first running the C preprocessor which concatenates all C files. We do a similar, but much more sophisticated form of preprocessing to combine all Elm modules into a single file so that it can be reduced with C-Reduce. We do this using Elm-reduce’s merge-module passes.

We ignore testing a naive approach with the DDmin implementations because they are included with C-Reduce in terms of passes.

To get the most direct comparison to what Elm-reduce does we ran C-Reduce on the webvim project with the ‘Map.!’ bug. After vendoring libraries and merging all modules into one file, `Main.elm`, we executed `c-reduce ./testscript Main.elm -not-c`. The results of this reduction can be seen in the first three columns of figure 4.1. The reduction took more than 8 days 24h a day on a low-tier virtual server[4].

Although we initially planned on doing this comparison for all combinations of bugs and projects, we had to reconsider, since it would have taken us approximately 10 weeks to test this and that was out of our time constraints.

[3]It is possible to run C-Reduce with multiple files, but during reduction, it will get rid of all directory structure of input files and saves files on top-level. So for example module `Update.Caret` and `View.Caret` would clash.

[4]1 CPU, 1GB RAM with 2 GB swap, SSD. Reads and writes should have been in-memory because reduction happens in the `/tmp` directory, which is mounted in tmpfs.

	webvim		we-connect (no vendoring)	
Pass	worked	failed	worked	failed
`pass_clex`	3790	681,495	321	113694
`pass_lines`	6208	461,202	1222	66724
`pass_balanced`	817	11,863	274	2317
`pass_peep`	60	6098	5	1354
`pass_ints`	48	744	2	72
`pass_indent`	3	8	0	0
`pass_blank`	1	0	1	0
Total	10,927	1,161,410	1825	184,161
Percentage	0.93%	99.07%	0.98%	99.02%
Size (factor from optimal)	768 (153.6x)		290 (58x)	

Figure 4.1.: This table summarizes the tested runs of a naive approach using the existing C-Reduce reducer with the `--not-c` flag. Size is measured in lines of code, the number in brackets is by what factor the resulting minimal test case is bigger than the optimal, minimal test case.

To save time, we changed the way we tested the C-Reduce reduction. However, this substantially changes the circumstances in which the reduction is performed in compared to reductions using Elm-reduce. Concretely, we

- reduced test case size by skipping the dependency vendoring
- do not reduce the project configuration `elm.json` file
- changed the test case to the 'we-connect' project, which is about a sixth the size of webvim

The reduction took about a day on the same server with. Its results are summarized in the last 2 columns of figure 4.1. Keep in mind that this does not speak for C-Reduce's performance in general, as – to our knowledge – it is the best reducer for C code currently available.

4.2.2. Elm-reduce

To get a feeling of what impact different projects and interestingness tests have on the reduction time and behavior of Elm-reduce, we ran it on all 9 combinations of project-bug combinations. Some combinations are manufactured: All 'Map.!' bug test cases happened in this form to users, which we consider them 'real-world', whereas all other bugs were implanted into the projects.

The results of these can be seen in figure 4.2. We also provide the optimal minimal test case in comparison to Elm-reduce's minimal test case. From the size column, we can see that Elm-reduce consistently produces the same test cases for the 'Map.!' and 'out-of-memory' bugs. There seems to be room for improvement in comparison to the optimal minimal test case, but this only due to newline formatting.

It does not consistently produce similar test cases for the 'out-of-bounds' bug, because sometimes the implanted functions have different amounts of parameters, causing some extra lines. Removing parameters of top-level functions is not a pass in Elm-reduce yet, as it is quite difficult to get right in a way that keeps well-typed-ness.

For a more detailed view of the reduction evolution, we take a look at figure 4.3, which shows all Elm-reduce reduction evolutions we ran in comparison. While in general, they all perform similarly, there are some noticeable differences:

- The Map.! bug results in smaller peak project sizes during reduction, because it always prevents a library from being vendored. This is the library from which our opaque, higher-kinded type is referred from. When it is vendored, the bug vanishes.
- Many reductions perform identical or almost identical for some iterations: The 'flatawesome out of bounds' and 'flatawesome out of memory' or the 'webvim out of bounds' and 'webvim out of memory'. The reason for this could be that the bugs were planted in the same function definitions, causing identical reduction up until function-body-specific passes.

You might have noticed that in figure 4.2, reducing webvim projects takes less test case invocations than reducing flatawesome projects, even though flatawesome only has about 2,000 lines of code, and webvim had about 15,000. We can also see this in figure 4.3: There is a huge flat reduction curve for all 'flatawesome' reductions between iteration 250 and 1250. During these iterations, Elm-reduce removes dead function definitions from the `elm-css` package, a library that replicates all CSS[5] properties as functions. The slope appears flat, because Elm-reduce removes function definitions one by one, and there a lot of small functions to remove.

[5]'Cascading Style Sheet'

Project	Bug	steps	runtime	size (LoC)	optimal	factor
webvim	Map.!	2094	64 min	14	8	1.75x
we-connect	Map.!	1674	36 min	14	8	1.75x
flatawesome	Map.!	2056	55 min	14	8	1.75x
webvim	out-of-bounds	1695	23 min	19	17	1.12x
we-connect	out-of-bounds	1751	47 min	21	17	1.24x
flatawesome	out-of-bounds	2485	47 min	20	17	1.18x
webvim	out-of-memory	1525	26 min	15	9	1.67x
we-connect	out-of-memory	1720	36 min	15	9	1.67x
flatawesome	out-of-memory	2183	64 min	15	9	1.67x

Figure 4.2.: This table compares the reductions of different projects with bugs implanted. 'Steps' are the number of interestingness test invocations, 'rumtime' the execution time on the same virtual server as the tests for C-Reduce, and 'size' the reduced project's size in formatted lines of code. The 'optimal' column is the size of what we believe are the optimal minimal test cases from section 4.1.1. The factor column gives a size factor comparison between optimal and actual test case. We evaluate test case quality using lines of code so that identifier length does not affect results.

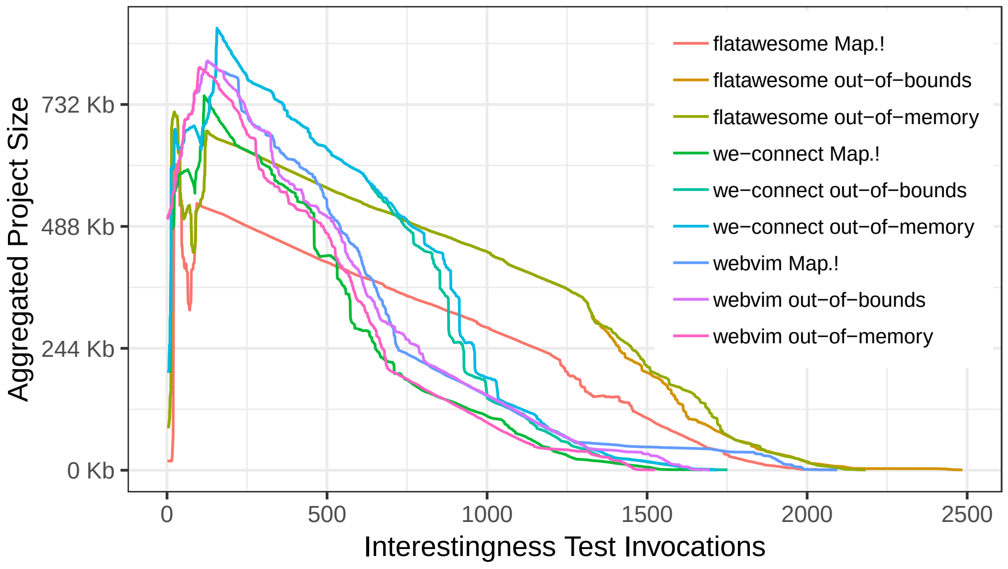


Figure 4.3.: This figure compares Elm-reduce reductions of all test project-bug combinations. For each reduction, it shows the test case size against the amount of interestingness test invocations.

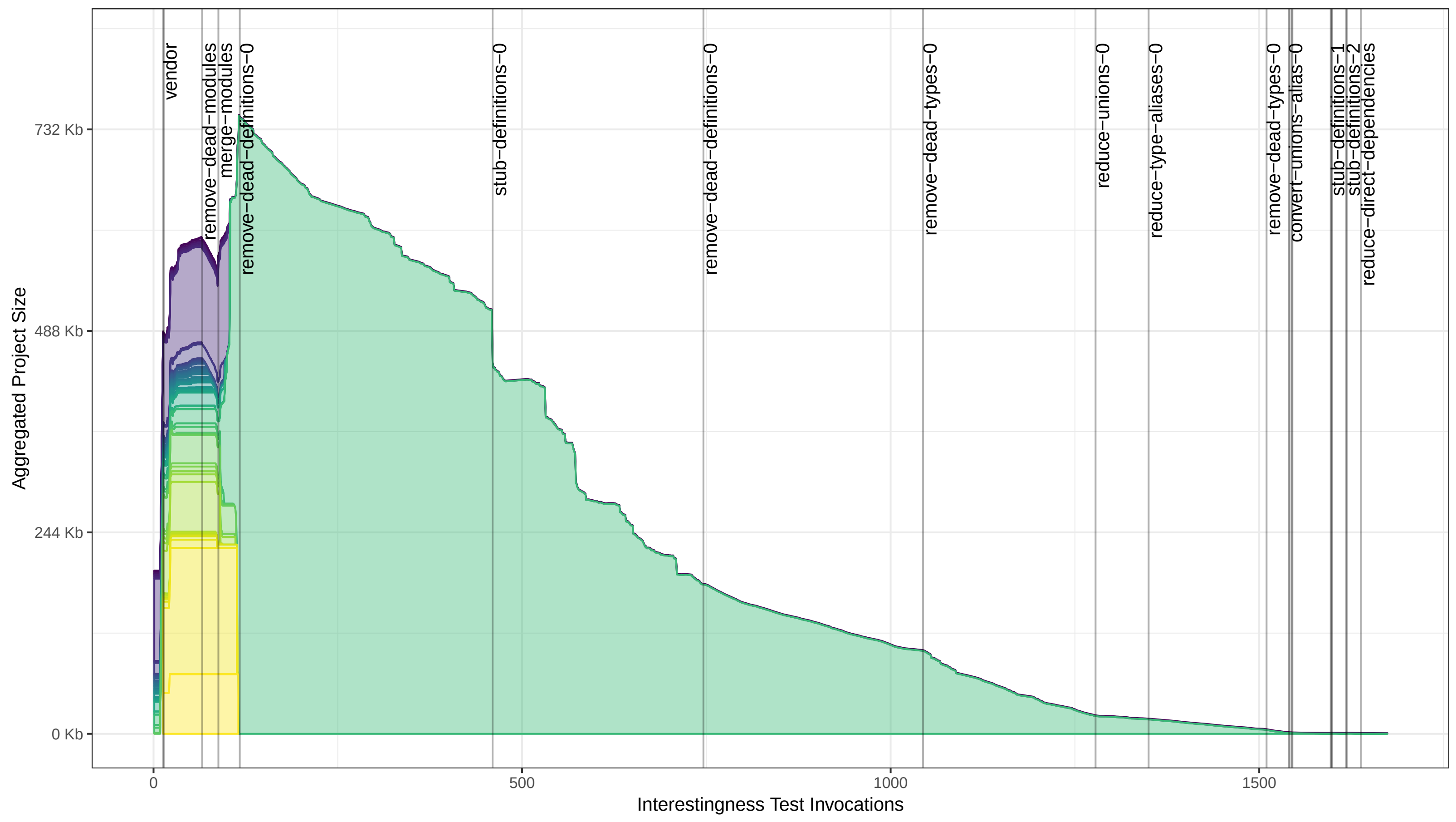


Figure 4.4.: Elm-reduce reduction progress for the 'we-connect' project and the 'Map.!' bug. Files are colored individually, `Main.elm` is green. The vertical bars mark changes in the passes used. The trailing number on some of the pass' names is the index of fixpoint iteration. More than 95% reduction is achieved within the first iteration (trailing 0), but the whole reduction takes 3 iterations to end up at a fixpoint. The resulting project size is 903 bytes with a 14 lines of code `Main.elm` file.

Figure 4.4 gives a very detailed look at what happens during the reduction: Project size dramatically increases from initial size to the peak due to two passes: Dependency vendoring copies all dependencies' source into the project and the merging modules pass increases identifier length. There is a small dent during the increase to the peak because the dead module removal pass deletes all vendored library modules that are never referenced.

At the peak, all modules were combined into the `Main.elm` file and from here on, all passes (except for dependency removal) are only applied to this file. Because there are still lots of vendored library functions never referenced, Elm-reduce spends 343 test executions removing dead functions. Then the next 286 test invocations all function bodies are replaced with a stub. In the graph, we can see huge reductions, when big function bodies are eliminated. The following dead definition removal pass is very linear because all functions removed have approximately the same size: Their body in most cases only consists of our stub.

At this point, there exist no more function definitions in our code base, except for a stubbed definition for `main`. Elm-reduce now narrows down the type declaration that causes the Map.! bug by removing all irrelevant type declarations and doing some inlining and reduction passes, as described in chapter 3.

After 1597 test invocations Elm-reduce applied all passes once and reduced project size by about 95% in comparison to its peak. It now proceeds with additional runs of all passes to find the fixpoint, and after 2 more runs of all passes, a fixpoint was found.

In some cases the second fixpoint iteration does significantly more work than in this example: In the reduction fo 'webvim', some modules can't be merged because of a complicated identifier shadowing problem that would require substantial implementation effort to account for. In this case Elm-reduce simply removes the modules in the second fixpoint iteration, because they are not referenced anymore. We believe that there are more such cases in which another fixpoint iteration can pick up new reduction opportunities.

5. Conclusion and Future Work

Our results show that general delta debugging approaches don't work, when test case data validity is sophisticated, as is the case with source code. 1-minimal test cases in terms of input characters are not satisfyingly close to the actual minimal test cases. We need language-specific reduction passes to replace deleted parts of input with, so we still have valid source code. All of Elm-reduce's destructive passes not only remove characters from source code but also replace them with language-specific constructs like pattern match wildcards or specific source code expressions.

5.1. Improving Performance

Elm-reduce's main metric was the quality of the resulting minimal reduction, but we didn't optimize regarding reduction performance. However, Looking at the reduction process closely we identified some possible performance improvements.

Careful Selection of next Reduction

Sometimes, Elm-reduce attempts to delete a function that can't be removed without failing the interestingness test, but when it moves on to deleting another function, it comes back to try deleting the previous function again. However, those contain the triggering expression for an interestingness test, so the interestingness test will always fail on these attempts. Because Elm-reduce repeatedly attempts reducing them after every successful reduce, this leads to between 2-3 times as much interestingness test invocations more than if we were more careful with which functions to reduce next.

Reducing in Batches

The delta debugging algorithms have good performance because they attempt to reduce bigger batches of data at once. Because we have a complex graph of dependencies between functions and type declarations, we have to be careful reducing only nodes that are not referred. Removing those frees up the nodes they referred to for reduction. We could try removing more function definitions at the same time without generating stale references by better analyzing the dependency graph between functions.

5.2. Isolating Failure-Inducing Input

In the Elm compiler issue tracker, bug reporters are asked to submit both a minimal test case that reproduces the bug as well as the minimal change that needs to be applied to the test case to stop it from reproducing the bug.

Finding this minimal change can also be automated by reducing both an interesting and uninteresting variant at the same time [4], trying to reduce the difference (‘delta’). For this, it would be necessary to have an additional return type for interestingness tests: unresolved.

5.3. Reducing Code Generation Bugs

We believe it is possible to even test complicated code-generation bugs. The big issue with code-gen bugs is that it is sometimes necessary to provide input data to the issue-generating bug, for example, a code-generation bug on a website might only trigger, when there a feature is accessed with complex user interaction.

All code paths that the user interaction touches can’t be reduced. Consider a code-gen bug that it is usually triggered by a particular browser action when clicking a button. It is impossible to remove the button from the source code, even though it would be possible to trigger the bug simply by running the action the button produces at website loading time.

However, if the user interaction were reduced in tandem with the source code, even bugs that were triggered by complex user interaction could reduce to minimal test cases. In Elm, it would be possible to record user interaction and replay them, because of its elegant runtime system design. Elm-reduce would then need more sophisticated reduction passes like beta reduction and inlining so that it would weave together test data with test case code.

5.4. Finding Compiler Bugs

Elm-reduce can only reduce compiler bugs that were already triggered in some project, but it can’t prevent compiler bugs before they happen to users. This could be solved by trying to find compiler bugs by compiling random, ‘fuzzed’ source code.

Fuzzed test cases often contain lots of irrelevant data, which could then be removed by Elm-reduce. These fuzzer-reducer combinations have been very successful at finding compiler bugs [9][10].

Bibliography

[1] E. Czaplicki and S. Chong, "Asynchronous functional reactive programming for guis," in *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16-19, 2013* (H. Boehm and C. Flanagan, eds.), pp. 411–422, ACM, 2013.

[2] D. Leijen, "Extensible records with scoped labels," in *Revised Selected Papers from the Sixth Symposium on Trends in Functional Programming, TFP 2005, Tallinn, Estonia, 23-24 September 2005.* (M. C. J. D. van Eekelen, ed.), vol. 6 of *Trends in Functional Programming*, pp. 179–194, Intellect, 2005.

[3] A. Zeller, "Yesterday, my program worked. today, it does not. why?," in *Software Engineering - ESEC/FSE'99, 7th European Software Engineering Conference, Held Jointly with the 7th ACM SIGSOFT Symposium on the Foundations of Software Engineering, Toulouse, France, September 1999, Proceedings* (O. Nierstrasz and M. Lemoine, eds.), vol. 1687 of *Lecture Notes in Computer Science*, pp. 253–267, Springer, 1999.

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

[5] G. Misherghi and Z. Su, "HDD: hierarchical delta debugging," in *28th International Conference on Software Engineering (ICSE 2006), Shanghai, China, May 20-28, 2006* (L. J. Osterweil, H. D. Rombach, and M. L. Soffa, eds.), pp. 142–151, ACM, 2006.

[6] J. Regehr, Y. Chen, P. Cuoq, E. Eide, C. Ellison, and X. Yang, "Test-case reduction for C compiler bugs," in *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '12, Beijing, China - June 11 - 16, 2012* (J. Vitek, H. Lin, and F. Tip, eds.), pp. 335–346, ACM, 2012.

[7] C. McBride and R. Paterson, "Applicative programming with effects," *J. Funct. Program.*, vol. 18, no. 1, pp. 1–13, 2008.

[8] T. M. Strößner, "Firmreduce: Automated test-case reduction for graph-based compilers," Sept. 2018.

[9] X. Yang, Y. Chen, E. Eide, and J. Regehr, "Finding and understanding bugs in C compilers," in *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA,*

USA, June 4-8, 2011 (M. W. Hall and D. A. Padua, eds.), pp. 283–294, ACM, 2011.

[10] G. Samuel Groß, “Fuzzilli.” `https://github.com/googleprojectzero/fuzzilli`, 2019.

Erklärung

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

Ort, Datum   Unterschrift

A. Appendix

A.1. Incrementally Reducing Graphs

Claim. *For all directed, acyclic graphs (DAGs) $G = (V, E)$ with $|V| > 1$ and exactly one source $v_s \in V$ there exists a vertex $v \in V$ with exactly one in-going edge $e \in E$ to v.*

Proof. Let G be a DAG with $|V| > 1$ and exactly one source $s \in V$.

$deg^-(v) \leq 1$, because there is only one vertex preceding it in the topological sort p.

$deg^-(v) > 0$, because if it were not, both $v_{p(1)} = v_s$ and $v_{p(2)}$ would be sources of the graph G.

Therefore $deg^-(v) = 1$ with a single edge e incoming to vertex v. □

Claim. *Let $G = (V, E)$ be a DAG with exactly one source v_s and $|V| > 1$. Let $e \in E$ be a single in-going edge that must exist according to the above claim. The edge-contracted graph $G' = G/e$ is a DAG and has exactly one source v_s.*

Proof. The resulting graph is a DAG, because edge contraction on simple graphs does not introduce loops.

It must have exactly one source v_s, because all other vertices v_i were reachable from v_s and all $v_i \in V'$ are also reachable by v_s:
Let p be the path that v_i could be reached from v_s. If the contracted edge e was part of p, then there exists a path p' that is obtained from contracting the edge e out of p that connects v_s to v_i in G'. If the contracted edge e was not part of p, then the same path connects v_s and v_i in G'. □

Claim. *A DAG with a single source v_s can be contracted with the above edge contractions to a graph containing only a single vertex.*

Proof. Either the graph $G = (V, E)$ is only a single vertex and we are done, or it has $|V| > 1$ and is contractible inductively with the above edge contractions, which always reduce graph size and either produce a DAG with a single source and either $|V| = 1$ or $|V| > 1$. □

A.2. Applicative Functor Laws for Reductions

A.2.1. Identity

Claim. $\forall vr.\ pure\ id <\!*\!> vr = vr$

Proof. Let $vr = Reductions\ v\ vs$, then

$$\begin{aligned}
& pure\ id <\!*\!> vr \\
=\ & Reductions\ id\ [\,] <\!*\!> vr \\
=\ & Reductions\ (id\ v) \\
& (map\ (\$\ v)\ [\,] +\!\!+\ map\ id\ vs) \\
=\ & Reductions\ v\ ([\,] +\!\!+\ map\ id\ vs) \\
=\ & Reductions\ v\ vs \\
=\ & vr
\end{aligned}$$

$\square$

A.2.2. Composition

Claim.

$$\begin{aligned}
& \forall ur, vr, wr. \\
& \quad pure\ (\circ) <\!*\!> ur <\!*\!> vr <\!*\!> wr \\
=\ & ur <\!*\!> (vr <\!*\!> wr)
\end{aligned}$$

Proof. Let

$$\begin{aligned}
ur &= Reductions\ u\ us \\
vr &= Reductions\ v\ vs \\
wr &= Reductions\ w\ ws
\end{aligned}$$

Then

$$
\begin{aligned}
&\ \mathit{pure}\ (\circ) <\!*\!> \mathit{ur} <\!*\!> \mathit{vr} <\!*\!> \mathit{wr}\\
&= \mathit{Reductions}\ (\circ)\ []\ <\!*\!> \mathit{ur} <\!*\!> \mathit{vr} <\!*\!> \mathit{wr}\\
&= \mathit{Reductions}\ ((\circ)\ u)\\
&\qquad (\mathit{map}\ (\$\ u)\ []\ +\!\!+\ \mathit{map}\ (\circ)\ \mathit{us})\\
&\qquad <\!*\!> \mathit{vr} <\!*\!> \mathit{wr}\\
&= \mathit{Reductions}\ ((\circ)\ u)\ (\mathit{map}\ (\circ)\ \mathit{us})\\
&\qquad <\!*\!> \mathit{vr} <\!*\!> \mathit{wr}\\
&= \mathit{Reductions}\ ((\circ)\ u\ v)\\
&\qquad (\mathit{map}\ (\$\ v)\ (\mathit{map}\ (\circ)\ \mathit{us})\\
&\qquad\ +\!\!+\ \mathit{map}\ ((\circ)\ u)\ \mathit{vs})\\
&\qquad <\!*\!> \mathit{wr}\\
&= \mathit{Reductions}\ ((\circ)\ u\ v)\\
&\qquad (\mathit{map}\ (\lambda g.\ (\circ)\ g\ v)\ \mathit{us}\\
&\qquad\ +\!\!+\ \mathit{map}\ (\lambda f.\ (\circ)\ u\ f)\ \mathit{vs})\\
&\qquad <\!*\!> \mathit{wr}\\
&= \mathit{Reductions}\ ((\circ)\ u\ v\ w)\\
&\qquad (\mathit{map}\ (\$\ w)\\
&\qquad\quad (\mathit{map}\ (\lambda g.\ (\circ)\ g\ v)\ \mathit{us}\\
&\qquad\quad\ +\!\!+\ \mathit{map}\ (\lambda f.\ (\circ)\ u\ f)\ \mathit{vs})\\
&\qquad\quad\ +\!\!+\ \mathit{map}\ ((\circ)\ u\ v)\ \mathit{ws}\\
&\qquad)\\
&= \mathit{Reductions}\ ((\circ)\ u\ v\ w)\\
&\qquad (\mathit{map}\ (\lambda g.\ (\circ)\ g\ v\ w)\ \mathit{us}\\
&\qquad\ +\!\!+\ \mathit{map}\ (\lambda f.\ (\circ)\ u\ f\ w)\ \mathit{vs}\\
&\qquad\ +\!\!+\ \mathit{map}\ (\lambda a.\ (\circ)\ u\ v\ a)\ \mathit{ws}\\
&\qquad)\\
&= \mathit{Reductions}\ (u\ (v\ w))\\
&\qquad (\mathit{map}\ (\lambda g.\ g\ (v\ w))\ \mathit{us}\\
&\qquad\ +\!\!+\ \mathit{map}\ (\lambda f.\ u\ (f\ w))\ \mathit{vs}\\
&\qquad\ +\!\!+\ \mathit{map}\ (\lambda a.\ u\ (v\ a))\ \mathit{ws}\\
&\qquad)\\
&= \mathit{Reductions}\ (u\ (v\ w))\\
&\qquad (\mathit{map}\ (\lambda g.\ g\ (v\ w))\ \mathit{us}\\
&\qquad\ +\!\!+\ \mathit{map}\ u\\
&\qquad\quad (\mathit{map}\ (\lambda f.\ f\ w)\ \mathit{vs}\\
&\qquad\quad\ +\!\!+\ \mathit{map}\ v\ \mathit{ws})\\
&\qquad)\\
&= \mathit{ur} <\!*\!>\\
&\quad \mathit{Reductions}\ (v\ w)\ (\mathit{map}\ (\lambda f.\ f\ w)\ \mathit{vs}\ +\!\!+\ \mathit{map}\ v\ \mathit{ws})\\
&= \mathit{ur} <\!*\!> (\mathit{vr} <\!*\!> \mathit{wr})
\end{aligned}
$$

□

A.2.3. Homomorphism

Claim. $\forall f, x.\ pure\ f <*> pure\ x = pure\ (f\ x)$

Proof.

$$\begin{aligned}
&pure\ f\ \ <*>\ \ pure\ x\\
&= Reductions\ f\ []\ \ <*>\ \ Reductions\ x\ []\\
&= Reductions\ (f\ x)\ (map\ (\$\ x)\ []\ \ +\!+\ \ map\ f\ [])\\
&= Reductions\ (f\ x)\ []\\
&= pure\ (f\ x)
\end{aligned}$$

□

A.2.4. Interchange

Claim. $\forall ur, x.\ ur <*> pure\ y = pure\ (\$\ y) <*> ur$

Proof. Let $ur = Reductions\ u\ us$, then

$$\begin{aligned}
&ur <*> pure\ y\\
&= ur <*> Reductions\ y\ []\\
&= Reductions\ (u\ y)\ (map\ (\$\ y)\ us +\!+ map\ u\ [])\\
&= Reductions\ ((\$\ y)\ u)\ (map\ (\$\ y)\ us)\\
&= Reductions\ ((\$\ y)\ u)\ (map\ (\$\ u)\ [] +\!+ map\ (\$\ y)\ us)\\
&= Reductions\ (\$\ y)\ [] <*> Reductions\ u\ us\\
&= pure\ (\$\ y) <*> ur
\end{aligned}$$

□

A.3. Associativity Law for Reductions

This law is required for Reductions to be a semigroup.

Claim. $\forall xr, yr, zr.\ xr <> (yr <> zr) = (xr <> yr) <> zr$

Proof. Let

$$\begin{aligned}
xr &= Reductions\ x\ xs\\
yr &= Reductions\ y\ ys\\
zr &= Reductions\ z\ zs
\end{aligned}$$

Then

$$\begin{aligned}
& xr <> (yr <> zr) \\
= {} & xr <> \mathit{Reductions}\ y\ (ys +\!\!+ zs) \\
= {} & \mathit{Reductions}\ x\ (xs +\!\!+ (ys +\!\!+ zs)) \\
= {} & \mathit{Reductions}\ x\ ((xs +\!\!+ ys) +\!\!+ zs) \\
= {} & (\mathit{Reductions}\ x\ (xs +\!\!+ ys)) <> \mathit{Reductions}\ z\ zs \\
= {} & (\mathit{Reductions}\ x\ xs <> \mathit{Reductions}\ y\ ys) <> zr \\
= {} & (xr <> yr) <> zr
\end{aligned}$$

$\square$