Jonathan Watson

Practical Dynamic Software Updating for Haskell

Computer Science Tripos – Part II

Queens' College

May 12, 2023

Declaration

I, Jonathan Watson of Queens' College, being a candidate for Part II of the Computer Science Tripos, hereby declare that this dissertation and the work described in it are my own work, unaided except as may be specified below, and that the dissertation does not contain material that has already been used to any substantial extent for a comparable purpose. In preparation of this dissertation I did not use text from AI-assisted platforms generating natural language answers to user queries, including but not limited to ChatGPT. I am content for my dissertation to be made available to the students and staff of the University.

Signed: Jonathan Watson

Date: May 12, 2023

Proforma

Candidate Number:	**2431C**
Project Title:	**Practical Dynamic Software Updating for Haskell**
Examination:	**Computer Science Tripos – Part II, 2023**
Word Count:	**11,993**[1]
Code Line Count:	**9,310**[2]
Project Originator:	**The Candidate**
Project Supervisor:	**Mistral Contrastin**

Original Aims of the Project

Dynamic software updating (DSU) involves patching programs while they are running, without losing state. This project aimed to implement a DSU system that can retrofit and update existing long-running Haskell programs, while minimising programmer effort. This involves implementing a DSU runtime, a library for defining type-driven state transformers, a plugin for injecting data from our runtime into programs, and a command line interface. As an extension, unloading versions of programs would be supported by our DSU runtime. The practicality of the DSU system would be evaluated by quantifying the number of updates from existing programs that can be performed.

Work Completed

The original success criteria were met. For a sample of existing Haskell programs, my DSU system could perform almost every modern commit as an update. I designed a framework for representing versions of programs and their updates with packages. I implemented a DSU runtime that loads these packages and interacts with them. I developed a flexible and type-safe library that allows state transformers to be both automatically generated and manually defined. I also implemented a type-checking plugin to safely inject runtime data, as well as a CLI that generates DSU runtimes and simplifies retrofitting existing programs by utilising patch files.

Special Difficulties

None.

[1] This word count was computed using `texcount`, configured to include tables.

[2] This code line count was computed with `cloc` using the command `cloc ./lowarn ./lowarn-evaluate --include-lang "haskell,c,bourne shell,python" --skip-uniqueness --json | jq '[.SUM.code,.SUM.comment] | add'`.

Contents

1 Introduction

Dynamic Software Updating (DSU) is a technique that involves patching programs while they are running, without losing their state. According to Tesone et al. [1], DSU is typically either used in production, for long-running applications, or in integrated development environments, for live coding systems.

In this dissertation, we explore how adding DSU to existing applications can be made practical. Therefore, we consider long-running applications rather than live coding systems, which inherently have DSU. Furthermore, we discuss how we can achieve this in the context of the Haskell programming language.

1.1 Practical DSU

Long-running applications, such as web servers, database management systems, and operating systems, are often hard to update. Restarting these applications can cause data loss and disruption, which can be costly when zero downtime deployment or high availability is required. Redundancy is typically used to mitigate these problems, but this is complex and expensive. DSU can present an easier solution.

One of the major disadvantages of DSU is that it has far-reaching implications for the design and architecture of programs that use it, as considerations must be made around how and when updates can occur. As a result, adding DSU to a program is often considered a design decision that must be made early in the development process. The motivation behind this project is to examine how existing programs can be retrofitted with DSU without the need for major design changes, providing the benefits of high availability without requiring significant programmer effort.

In Section 2.1, we discuss existing DSU systems and the approaches they use. From this, we base our work on a specific approach implemented in *Kitsune* and *Rubah*, two DSU systems that have successfully updated several long-running programs while preserving both safety and performance. These systems are characterised by a simple yet safe approach to determining when an update can occur, along with very low overheads.

1.2 DSU for Haskell

Haskell is a purely functional programming language, known for lazy evaluation and strong type safety. Its safety and tooling have made it a popular choice for implementing programs that use DSU.

Many DSU systems for live coding have been implemented with Haskell, as it can be used in GHCi, a robust interactive environment included with the Glasgow Haskell Compiler (GHC). Examples of these live coding systems include McLean's *TidalCycles* [2] and Bärenz's *essence-of-live-coding* [3].

Some systems also exist for implementing DSU for long-running Haskell applications. However, these systems are typically designed to update specially-selected components of programs, rather than entire programs. They are also more simple and generic than our

system, which means more programmer effort is required to use them. More [4] proposes an approach that uses *Cloud Haskell*, a framework for writing distributed applications in the style of the Erlang programming language, to update a running program by cloning its process. This system builds upon Stewart's *plugins* library [5], which allows DSU to be implemented by dynamically linking extensions that are type-checked at runtime. An early implementation of DSU for Haskell was introduced by Peterson et al. [6] for the Hugs bytecode interpreter, although this system is restrictive as it does not consider changes to the type of the state.

Our project involves applying an approach that has previously been successful for C and Java, in the hopes that it is as effective in Haskell. While many parts of this approach only rely on properties of long-running programs, we must make some changes for Haskell. For example, many programs that use DSU store the data from the DSU system in global variables, but these are unsafe and discouraged in Haskell. We therefore implement a compiler plugin that provides a safer form of global variables. Haskell's type system also results in more data needing to be transformed on every update, so a significant part of our work allows state transformers to be automatically defined in a more flexible and safe manner.

1.3 Project summary and key results

In this project, we implement and evaluate a DSU system for Haskell, called *Lowarn*, based on the approach found in Kitsune and Rubah. In particular, we:

- Develop a command line interface for running programs that use Lowarn and retrofitting existing programs easily (Section 3.2).
- Implement a *DSU runtime* that loads code, handles state and listens for updates (Section 3.3).
- Implement a compiler plugin that allows data from the DSU runtime to be injected into programs without much code modification or danger (Section 3.4).
- Introduce a flexible and safe library for implementing transformers between values of different types, using advanced features of Haskell's type system (Section 3.5).
- Apply Lowarn to 237 versions of three long-running programs, along with updates between all but two pairs of adjacent versions, demonstrating the flexibility of Lowarn (Section 4.1).
- Show that the amount of code modification required to retrofit existing programs with Lowarn is low and evaluate the overhead of retrofitting existing programs with Lowarn, finding that there is a small impact on speed and an impact on memory usage of over 200 MiB (Section 4.3, Section 4.4).

2 Preparation

We begin by discussing existing DSU systems, including two that we base our system on, in Section 2.1. The subsequent two sections describe background material required to understand the remainder of this dissertation. Section 2.2 covers modern developer tooling for Haskell and its applications to DSU, while Section 2.3 describes special features of the type system that we make use of. Following this, we are able to establish the requirements of our DSU system in Section 2.4. We then discuss professional practices in Section 2.6 and Section 2.7.

2.1 DSU Systems

Existing DSU systems can be evaluated in several ways. In this section, we discuss approaches used by existing DSU systems and how they compare. We then describe how an approach used by two DSU systems, Kitsune and Rubah, can be used in our project.

2.1.1 Timing Restrictions

There are many existing DSU systems, each differing in terms of safety guarantees and ease-of-use. The most commonly used form of safety, according to Ahmed et al. [7], is type safety. For an update to be type-safe, it must not be able to cause code to operate on data that is not of the correct type. To provide type safety, DSU systems typically employ *timing restrictions* that determine when updates can take place.

Timing restrictions are either implemented manually or automatically. In the former case, the programmer manually identifies update points, locations in the code where it is safe to perform an update. In the latter case, we infer when an update can occur. This can use dynamic or static analysis and typically involves determining if we have a given safety property. According to Hayden et al. [8], most DSU systems that use automatic timing restrictions either use *activeness safety* or *con-freeness safety*.

Systems like *OPUS*, by Altekar et al. [9], enforce that we have activeness safety when updates are performed. We have activeness safety if and only if there are no functions on the call stack that will be modified in the next update. Systems like *Ginseng*, by Neamtiu et al. [10], automatically enforce con-freeness safety, where updates can only take place at locations where updated types can not be accessed concretely.

By evaluating timing restrictions that use activeness safety and con-freeness safety alongside those that use manual update point identification, Hayden et al. [8] find that the manual approach, following simple design patterns, is the most effective. In particular, this approach requires the least programmer effort and results in the least failures. Two DSU systems that use this approach are described in Section 2.1.2.

2.1.2 Kitsune and Rubah

Kitsune for C, by Hayden et al. [11], and *Rubah* for Java, by Pina et al. [12], use manual timing restrictions following simple design patterns. These systems have successfully

updated several long-running, open source programs. Both systems impose no noticeable performed overhead on typical execution, require little programmer effort, and are flexible enough to allow almost all updates.

We describe the design of Kitsune, but Rubah works similarly. A long-running program must contain some form of event loop in its control flow, as shown in Figure 2.1. A program is considered *quiescent* at the start of an event loop. These papers define that a program is quiescent when state that is relevant to updates is not being modified. This property allows us to safely update a long-running program at the start of its event loop, as shown in Figure 2.2.

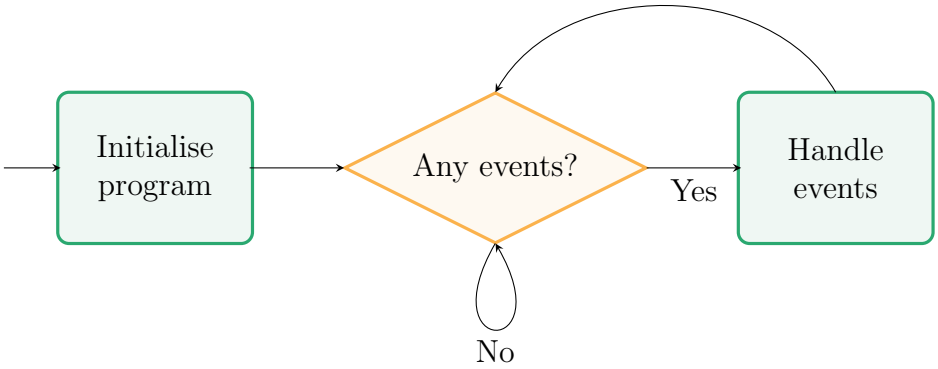


Figure 2.1: The control flow of a long-running program.

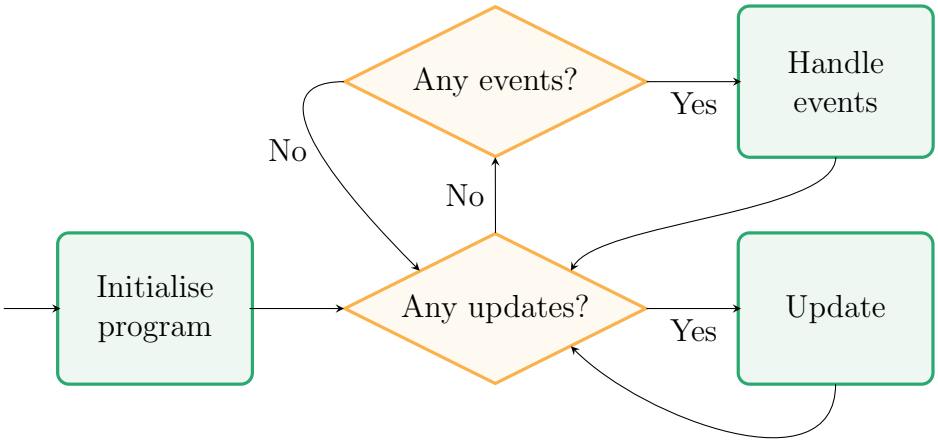


Figure 2.2: The control flow of a long-running program with an update point.

In Kitsune, updates are performed by loading in a new version of the program with dynamic linking and giving it the state of the previous version of the program. However, the new version of the program may represent state differently, so the state of the previous version of the program is transformed. These state transformations are specified in a language named xfgen that allows transformations between specific types to be described using C syntax with extra variables.

Once the new version of the program is loaded, it can return to the point where the previous version ended by making use of information provided by the DSU runtime system about the last update.

2.2 Tooling for Haskell

A practical DSU system should integrate with and make use of the existing tooling of the language it updates. The majority of modern Haskell programs are compiled with the *Glasgow Haskell Compiler* (GHC) and built with a build system like *Cabal*[1], *Stack*[2], or *Nix*[3]. This section describes these tools and how they can be used to facilitate DSU.

2.2.1 GHC

Haskell has multiple implementations that are standardised by the 2010 Haskell language report by Marlow et al. [13]. However, the Glasgow Haskell Compiler (GHC) is the only implementation that is widely used.

Like many existing Haskell projects, our work uses features that only work with GHC. One reason for this is that we need to interface with the specific compiler to load code at runtime. Using GHC also allows us to use some of its language extensions to implement metaprogramming features.

Our project uses version 9.2.7 of GHC. We need the version of GHC to be relatively modern such that it is compatible with many versions of modern programs that we update, but we also need it to be old enough for it to be known to not include major regressions. The GHC 9.2 series is 18 months old, at the time of writing.

2.2.2 Runtime Loading

DSU requires that new code is loaded at runtime. Typically, this is done by either interpreting byte-code or using *dynamic linking*. GHC supports both approaches, but, according to the GHC user's guide[4], byte-code is roughly 10 times slower than compiled code and does not support certain language extensions. Therefore, we make use of dynamic linking.

Use of GHCi, the interactive environment included in GHC, is notably widespread. As a result, GHC has good support for dynamic linking. GHC can either use a custom linker built into its runtime, or the system linker. The former linker, sometimes known as the GHCi linker, supports many operating systems and is unusual in that it loads static object files at runtime. The latter linker is supported on fewer operating systems, links shared object files, and cannot unload code. It is often more stable as its implementation is more simple.

The choice of linker involves many trade-offs, including ones that cannot be known without testing. Therefore, our DSU runtime system can make use of either linker. We end up using the system linker in our evaluation, as it has better support for the C foreign function interface.

[1] `https://www.haskell.org/cabal/`. Retrieved May 07, 2023.

[2] `https://haskellstack.org`. Retrieved May 07, 2023.

[3] `https://nixos.org`. Retrieved May 07, 2023.

[4] `https://downloads.haskell.org/~ghc/9.2.7/docs/html/users_guide/ghci.html`. Retrieved May 07, 2023.

2.2.3 Build Systems

For our DSU system to be practical, we need it to easily integrate with existing Haskell programs. This requires compatibility with standard project structures and the build systems used for them. Our DSU system is designed to work with *Cabal*, a build system used by almost every Haskell project, either directly or indirectly.

Cabal uses packages to organise and distribute code. GHC does have a notion of packages, but these types of package are mainly used to organise compiled libraries and are rarely registered manually. Cabal's packages instead contain source code and can produce executables, GHC packages, and operating system packages. In the Haskell ecosystem, Cabal packages are the main *unit of distribution*, meaning that they can be distributed independently in either source or binary form. As a result, we can easily reason about Haskell programs as packages. A key innovation of our DSU system is that it loads packages, rather than modules. This solves the problem of updating programs consisting of multiple modules, encountered when using existing DSU systems for Haskell that only update one module at once.

Other build systems that are widely used for Haskell include Stack and Nix. As both these systems tend to use Cabal under the hood, it is sufficient for our DSU system to only work with Cabal. However, we provide support for Stack regardless, as its approach to sandboxing makes linking easier than with Cabal.

2.3 The GHC Type System

In this section, we describe background material on the type system of GHC, the *de-facto* compiler for Haskell. We explore how GHC's type system works and how we use it to make implementing type-driven state transformations and runtime data injection safer and easier.

2.3.1 Types and Kinds

Our type-driven state transformation library generates state transformers between values of given datatypes by representing the values with advanced types, introduced in Section 2.3.5. In this subsection we describe *kinds*, used to reason about these types.

Haskell is known to have a rich static[5] type system. Inhabited types are types that have values (or $\perp$, which denotes a non-terminating computation). Type constructors are types[6] that can be given a (possibly zero) number of type arguments to produce a type. A non-nullary type constructors is an example of a type that is uninhabited.

A kind can be thought of as the type of a type. Most[7] inhabited types have the kind `Type` and every n-ary type constructor has the kind κ_0 `->` $\ldots$ `->` κ_n, where each κ_i is a kind.

[6] In this dissertation, we follow the convention of using "type" as an umbrella term for inhabited types and type expressions such as type constructors, following the terminology of the GHC user's guide.

[6] As described by Peyton Jones et al. [14], using reflection to generate runtime representations of types can enable a limited form of dynamic typing.

[7] *Unlifted types*, introduced by certain GHC extensions, are inhabited, but have the kind `#`. These types are strict and do not have $\perp$. We do not explicitly use unlifted types in our implementation.

For example, the declaration `data Maybe a = Just a | Nothing` introduces an unary type constructor `Maybe` and *data constructors* `Just` and `Nothing`. The type `Maybe Int` is inhabited (by $\perp$ and values like `Just 1`) and has kind `Type`, while the type `Maybe` is uninhabited and has the kind `Type -> Type`.

Promoted datatypes, introduced by Yorgey et al. [15], allow data to exist on the type level. With certain GHC extensions, the previous declaration of `Maybe a` also results in a kind `Maybe`, as well as type constructors `'Just` and `'Nothing`. These type constructors are disambiguated from the data constructors with a tick `'` that can be omitted when there is no ambiguity. `'Just` is an unary type constructor of the kind `a -> Maybe a`, while `'Nothing` is a nullary type constructor of the kind `Maybe a`. In these kinds, we see a variable `a` that matches all types of kind `Type`.

The work of Yorgey et al. [15] also introduces *kind polymorphism*, where kinds can include variables that can be substituted with all kinds. Weirich et al. [16] extend kind polymorphism by treating types and kinds as the same thing. Crucially, we let `Type` have the kind `Type`. As a result of this work, the previously described type `Maybe` and kind `Maybe` are in fact the same and both have the kind `Type -> Type`.

2.3.2 Type Classes

Our type-driven state transformation library provides a function that transforms state from one type to another. This function must use ad-hoc polymorphism, as its implementation must depend on the types of its input and output states. Haskell supports ad-hoc polymorphism with *type classes*. Usually, we can think of a type class as a function from types to values.

A type class includes a type constructor, a set of associated operations called methods, and a context that describes constraints that must hold. An *instance* of a type class associates the type constructor, given arguments of certain types, with implementations of the methods. When we use a method, we use an implementation from an instance with matching types.

Listing 1 shows an abridged declaration and instance of GHC's `Ord` type class, used to provide methods on types where we have a total order. The type constructor is `Ord` and the context is `Eq a`. This context declares that `Eq` is a *superclass* of `Ord` such that we have an instance `Ord a` only if we have an instance `Eq a` for the same type `a`.

This example also demonstrates *derived instances* that are automatically generated for datatypes specified with `deriving` syntax. It also includes a *minimal complete definition* pragma. This allows an instance to be declared by giving certain subsets of the methods, with default implementations being used to determine the other methods.

As described by Peyton Jones et al. [17], Haskell's conservatively-designed type class system can be extended in many ways. In our implementation, we use GHC extensions that allow type classes to have multiple parameters, more complex constraints, and overlapping instances. We also use type classes to do metaprogramming. This is possible due to type classes having parameters that can be determined from other other parameters, using either *functional dependencies* or *associated type families*, with associated type families being the favoured mechanism.

```
class Eq a => Ord a where
  {-# MINIMAL compare | (<=) #-}
  compare              :: a -> a -> Ordering
  (<), (<=), (>), (>=) :: a -> a -> Bool
  max, min             :: a -> a -> a

  compare x y = if x == y then EQ else if x <= y then LT else GT
  x < y       = case compare x y of { LT -> True;  _ -> False }
  x <= y      = case compare x y of { GT -> False;  _ -> True }
              ⋮

data AOrB = A | B deriving Eq

instance Ord AOrB where { B <= A = False; _ <= _ = True }
```

Listing 1: The `Ord` type class.

2.3.3 Type Families

The type-level data that we use in our type-driven state transformation library must be manipulated such that we can reason about it. We use *type families* for this purpose.

A type (synonym) family can be seen as a function that maps types to types and is either *open* or *closed*. A closed type family declaration contains all of its mappings. These mappings are checked sequentially, and can overlap in certain ways. Listing 2 shows the closed `Equals` type family, used in our implementation to perform type-level comparisons. It gives type-level data of the Boolean kind.

```
type family Equals a b :: Bool where
  Equals a a = 'True
  Equals a b = 'False
```

Listing 2: The `Equals` type family.

Open type families act like type classes, in that new instances can be added anywhere. Therefore, we are not able to define an order on the mappings of an open type family, like we can in the definition of `Equals`. Open type families can be included in type classes, where they are known as *associated types*. We use associated types extensively to infer type parameters of type classes from other type parameters.

2.3.4 Constraint Solving

In Section 3.4, we extend GHC's *constraint solver*. This is the part of GHC that type-checks programs. We do this to automatically add instances for type classes that have methods for injecting data into existing programs.

As summarised by Peyton Jones [18], GHC's approach to type checking somewhat resembles Algorithm W. This is an algorithm, described by Milner [19], that performs

type inference for Damas-Hindley-Milner type systems by introducing and solving equality constraints on the fly. GHC extends this approach by deferring constraint solving and introducing two new types of constraint.

From a Haskell source program, GHC attempts to generate an equivalent program in Core, a simplified intermediate language where all type information is present. This process is called elaboration and the resulting Core program is called the elaborated program. Elaboration acts as GHC's type-checking mechanism.

As shown in Figure 2.3, elaboration converts a Haskell source program to an elaborated program containing holes for unknown types and terms. We also get a set of constraints that are solved to fill these holes. Some of these constraints are the equality constraints used to unify parametrically-quantified type variables in Algorithm W. We also have constraints for type class instances, implications (used for introducing constraints in existentially quantified types), and more advanced type equalities.

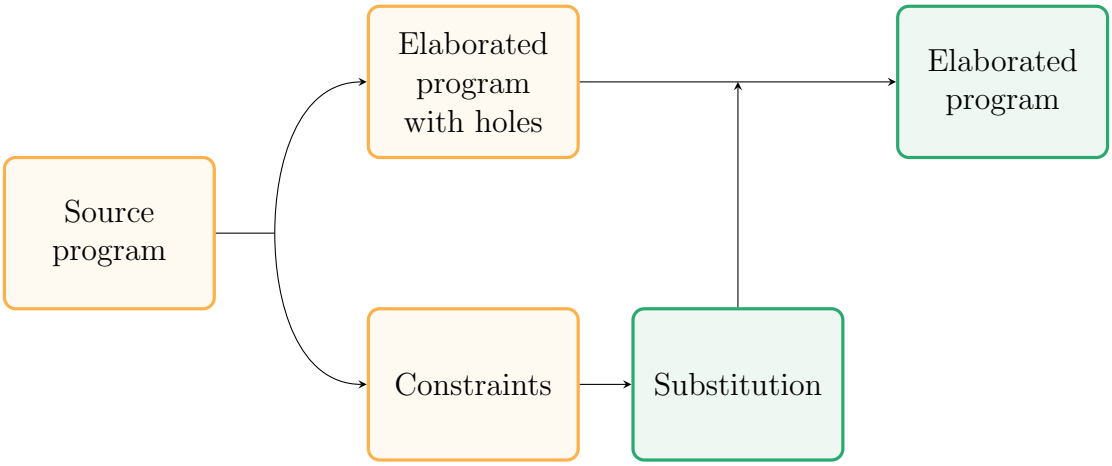


Figure 2.3: The GHC elaboration pipeline.

Internally, an instance of a type class is represented by GHC as a record known as a dictionary. This record contains an implementation of each method in the type class. For example, the `Ord AOrB` instance shown in Listing 1 results in a record containing each of the seven methods of `Ord`. In elaboration, a use of a method belonging to a type class results in this method being extracted from an unknown dictionary. This dictionary corresponds to a constraint that encodes the type class alongside its potentially unknown type parameters. The unknown type parameters are unification variables that can also appear in other constraints resulting from the use of the method.

Constraints are solved by the constraint solver with *evidence*, used to form a substitution that fills the holes in the elaborated program. An equality constraint is solved by providing a *coercion* as evidence, where a coercion is a type found in Core's language that proves that two types are equal. A type class constraint is solved by providing a dictionary corresponding to the instance as evidence.

Throughout constraint solving, GHC maintains sets of *given* and *wanted* constraints. Given constraints are constraints that are known to have evidence, while wanted constraints are constraints that we need evidence for. Given constraints are introduced by implication constraints and are not particularly relevant to our project. Constraint solving entails iteratively solving wanted constraints and introducing new given and wanted constraints until no wanted constraints remain, without backtracking.

We make use of GHC's *typechecker plugins*[8]. These plugins facilitate metaprogramming by exposing the sets of given and wanted constraints at certain points and allowing the programmer to provide custom evidence.

2.3.5 Generic Programming

In this subsection, we introduce *generic programming*, used to define state transformers for arbitrary datatypes.

Datatype-generic programming, also known as generic programming[9], lets us write functions that operate on arbitrary datatypes by representing each datatype in a standard form based on structure. The form of generic programming currently found in GHC is introduced by Magalhães et al. [20] with the `Generic` type class.

When we can derive an instance of `Generic a` for datatype `a`, we can convert values of type `a` to and from an associated type (see Section 2.3.3) `Rep a`. This type is constructed from six datatypes that can fully encode the datatype's representation and metadata. A type class that has instances for these six types can act upon all derived `Rep a` types.

We make use of the `generics-sop` library, introduced by de Vries and Löh [21]. This library defines an alternative `Generic` type class and `Rep a` associated type to represent a datatype as a *sum of products*, as shown in Listing 3.

```haskell
data Tree a = Leaf a | Branch (Tree a) (Tree a)

type instance Code (Tree a) = '['[a], '[Tree a, Tree a]]

type Rep (Tree a) = SOP I (Code (Tree a))

exampleLeafRep, exampleBranchRep :: Rep (Tree Int)
exampleLeafRep   =    Z (1 :* Nil)
exampleBranchRep = S (Z (Leaf 0 :* Leaf 1 :* Nil))

-- from exampleLeafRep   = Leaf 1
-- from exampleBranchRep = Branch (Leaf 0) (Leaf 1)
```

Listing 3: Sum of products representation of `Tree a`.

Sum of products (SOPs) are of the type `SOP f xss`, where `xss` is a type-level list of type-level lists of types of kind `k`, and `f` is an unary type constructor, or *functor*, that maps types of kind `k` to types of kind `Type`. We use the functor `f` to produce inhabited types from the arbitrary types in `xss`. In the context of SOP representations of datatypes, `k` is `Type`, `f` is the identity functor `I`, and `xss` is `Code a` for some datatype `a`.

The unique kind of the SOP type is similar to those of the sum and product types, `NS f xs` and `NP f xs`, although `xs` is 1-dimensional rather than 2-dimensional. The sum type is used to represent a choice of constructors and is represented with Peano-style

[8] `https://downloads.haskell.org/~ghc/9.2.7/docs/html/users_guide/extending_ghc.html`. Retrieved May 10, 2023.

[9] Generics, a feature of Java that adds parametric polymorphism, has no relation to datatype-generic programming.

natural numbers. The product type is used to represent a sequence of a constructor's fields and is represented with a heterogeneous list. `SOP f xss` is built from these types as `NS (NP f) xss`.

2.4 Requirements Analysis

Our main goal in this project is to allow many updates for existing long-running Haskell programs to be performed at runtime with minimal programmer effort. In order to support any form of updating, we need a DSU runtime system that can load Haskell code. To match the standard unit of distribution of existing Haskell projects, we require that this runtime system loads packages.

The constraints and requirements of our runtime system greatly influence the design of the whole DSU system. To ensure that we meet our goal of producing a practical DSU system, we also identify further core requirements that reduce boilerplate introduced by these design decisions. We also require a command line interface such that running and evaluating our code is practical.

Our core requirements are then as follows:

Runtime system. A DSU runtime system that can load versions and updates of a program into memory, as packages.

Type-driven state transformations. A library that allows state transformations to be declared in a type-driven manner. State transformations should be automatically generated for similar types. Custom state transformers for types that are not similar should be definable and integrated with automatic state transformer generation.

Runtime data injection. A system that allows data belonging to the runtime to be injected into programs and accessed at any point.

Command line interface. A command line interface (CLI) for organising projects that use our DSU system. Projects should be configurable such that the CLI can set options of the runtime. The CLI should be able to detect versions and updates and automatically load them. It should also provide utilities for retrofitting existing programs.

Our optional extension requirements are as follows:

Code unloading. Extensions to the DSU runtime system that allow loaded code to be unloaded once it is no longer required.

Label API. Extensions to the DSU runtime system that allow labels to be assigned to update points. After an update is performed, the following version of the program should be able to query the runtime data to find any labels associated with the update point where the previous version of the program ended.

2.5 Software Development Model

The core requirements described in Section 2.4 can be implemented as mostly independent modules or packages. A key advantage of this is that we can independently test and evaluate each feature throughout development. This approach is well-suited by the spiral development model, introduced by Boehm [22]. This involves successive waterfall iterations, consisting of design, implementation, and evaluation. The largest and highest-risk iterations are performed first, with subsequent iterations, informed by previous evaluation steps, being tighter and lower-risk.

Figure 2.4 shows the phases of the spiral model in our project.

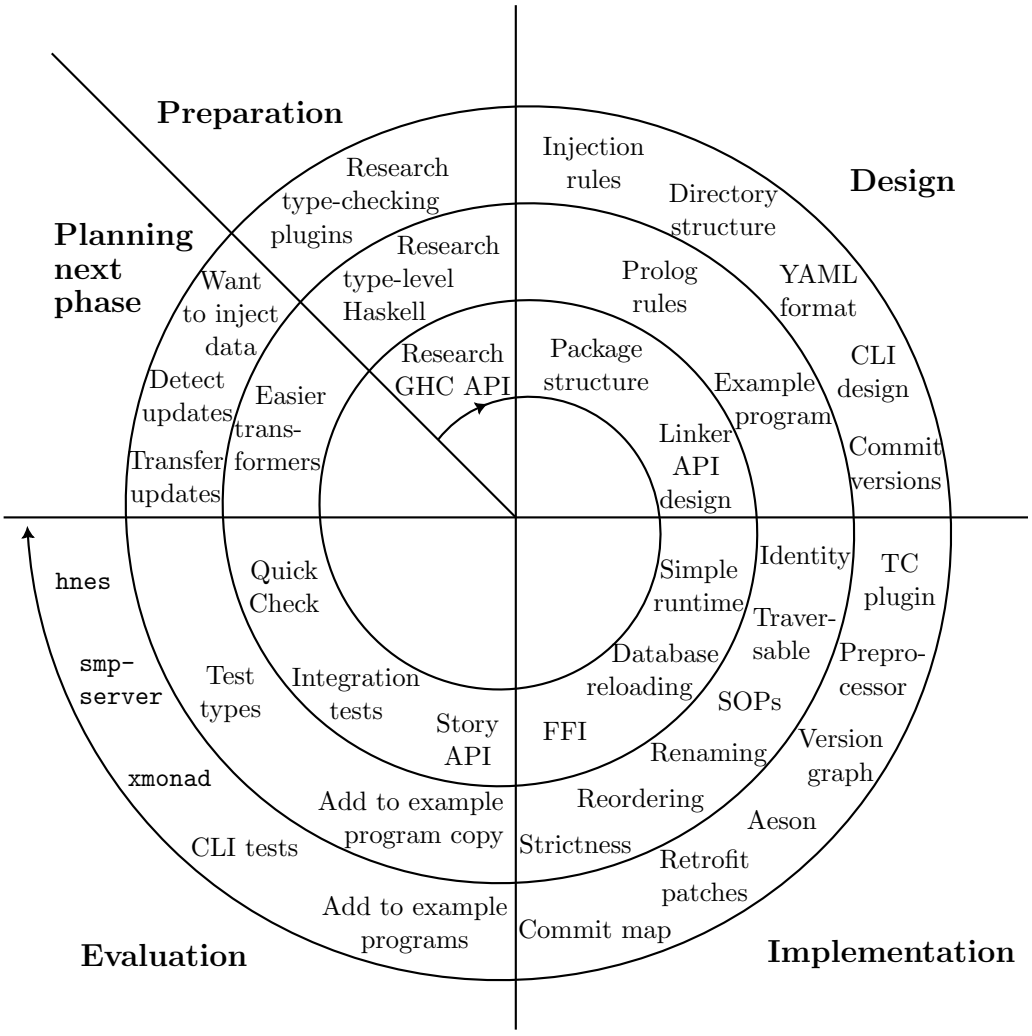


Figure 2.4: Spiral model diagram showing the project's three phases of development.

2.6 Development Tools

Our project can use either Stack or Cabal as a build system. The code has been formatted with the Ormolu code formatter[10] and checked with HLint[11].

[10] `https://github.com/tweag/ormolu`. Retrieved March 30, 2023.
[11] `https://github.com/ndmitchell/hlint`. Retrieved March 30, 2023.

We have 203 tests embedded in the documentation, run with `cabal-docspec`[12], and a further 105 tests found in a test suite that uses the Tasty testing framework[13]. It contains property tests, using `tasty-quickcheck`[13], as well integration tests that are validated using `tasty-golden`[14]. In total, we have **308 tests**. By forking a version of Dekking[15], a coverage reporting tool for Haskell, I found that the tests reach **84% coverage** for non-testing libraries.

Our main repository consists of multiple packages that have fully documented APIs. The documentation can be generated using the Haddock documentation tool[16].

For source control, we use Git, hosted on GitHub, making use of pull requests and issues. For CI, we use GitHub actions. On every commit on every branch, this checks the formatting with Ormolu, runs the tests, and generates the documentation.

2.7 Licensing

Our codebase consists of a repository containing several packages composing our DSU system, a repository containing code used for evaluation, and three repositories containing updates to existing Haskell projects.

I licensed our first two repositories under the permissive MIT licence such that they can be easily integrated with other Haskell projects. They include references to many other open source Haskell projects with permissive licences. As these are only references, their licences do not restrict the distribution of our source code. However, redistributions of the package's binaries would include the binaries of these dependencies, so additional copyright notices and licences, summarised in Appendix A, would need to be included with these.

The three repositories used for retrofitted programs are licensed under the same licences as the original programs, satisfying source code redistribution conditions. Two are licensed under BSD 3-Clause and one is licensed under AGPL-3.0-only.

Dekking, a coverage tool referenced in our main repository, has a custom licence[17]. Our repository meets its conditions. The tool is also only used in development.

2.8 Starting Point

I had no experience implementing DSU systems. I knew the basics of Haskell, having completed a Functional Programming in Haskell course in 2020, but I had no experience with real-world Haskell programs or advanced features that allow type-level programming. I also had no experience with Template Haskell or writing GHC plugins. I have practical experience with functional programming, having worked in industry with the OCaml programming language.

[12] `https://github.com/phadej/cabal-extras/tree/master/cabal-docspec`. Retrieved March 30, 2023.

[13] `https://github.com/UnkindPartition/tasty`. Retrieved March 30, 2023.

[14] `https://github.com/UnkindPartition/tasty-golden`. Retrieved March 30, 2023.

[15] `https://github.com/NorfairKing/dekking`. Retrieved April 03, 2023.

[16] `https://github.com/haskell/haddock`. Retrieved March 30, 2023.

[17] `https://github.com/NorfairKing/dekking/blob/master/LICENSE.md`. Retrieved May 07, 2023.

Prior to starting, I read literature on Ginseng, Kitsune, and Rubah DSU systems. I had some general knowledge of the low level concepts of intermediate representations and data representation from Compiler Construction.

2.9 Summary

We introduced multiple existing DSU systems, including Kitsune and Rubah. We then discussed the various components of Haskell, GHC, and build tools that can be used to implement DSU. We also justified decisions on requirements, software engineering methodology, development tools, and licensing.

3 Implementation

In this chapter, we discuss the design of our DSU system, known as Lowarn, beginning with an overview in Section 3.1. We discuss the CLI in Section 3.2, the DSU runtime in Section 3.3, runtime data injection in Section 3.4, and type-driven state transformations in Section 3.3.

3.1 Lowarn

Figure 3.1 depicts the phases of running and updating a program that supports Lowarn, using the command line interface. Each version of the program is represented by a package and each update of the program is represented by a package that depends on the version it updates to. However, version packages do not need to depend on update packages. As a result, we can load one version of a program without needing any future versions or updates to be loaded.

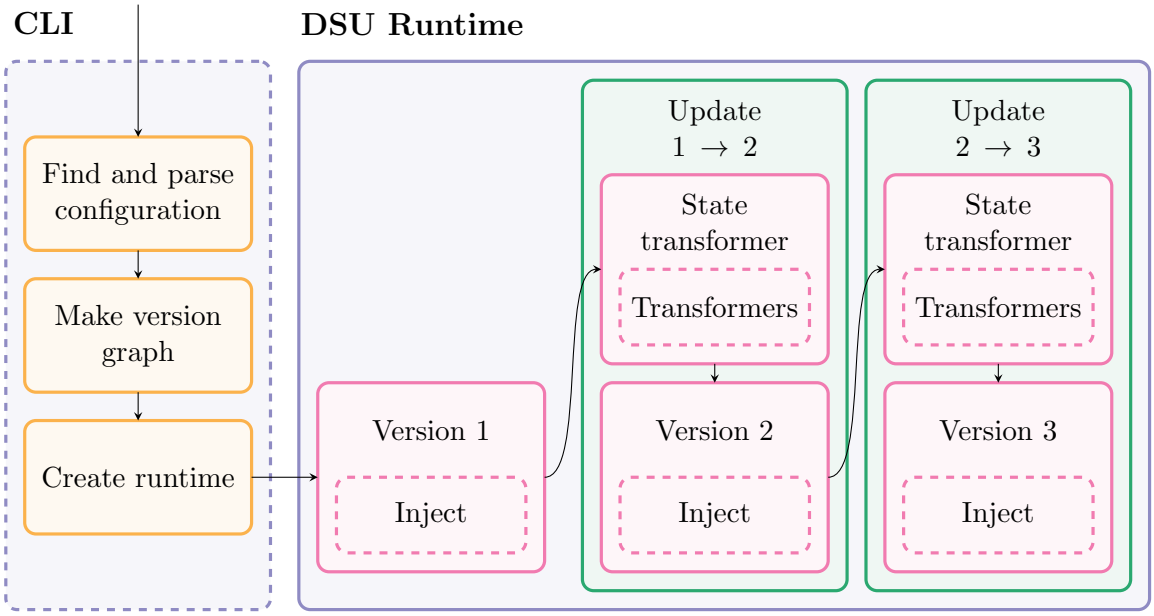


Figure 3.1: The Lowarn pipeline.

The pipeline begins with our CLI, explained in Section 3.2. It finds a configuration file for the program being updated and forms a graph consisting of its versions and updates. Then, it creates a DSU runtime, introduced in Section 3.3. This is a configurable system that runs alongside the program, handling code loading, state management, and listening for new updates.

The DSU runtime loads a version by calling a function in its package called its *entry point*. This function is given *runtime data*, used to acquire the state of the previous version of the program, if one exists, as well as to check if an update should occur. In Section 3.4, we introduce a compiler plugin that allows this data to be safely stored globally in a program without much code modification.

An update carries state from one version of a program to the next through a state transformer. This converts state represented with types found in the previous version of the program into state represented with types found in the next version of the program. Manually defining state transformers for every pair of types between two versions of a program can be tedious. In Section 3.5, we introduce our type-driven state transformation library, allowing state transformers to be defined automatically.

3.1.1 Package structure

Our project is split into multiple packages that depend on each other, as shown in Figure 3.2.

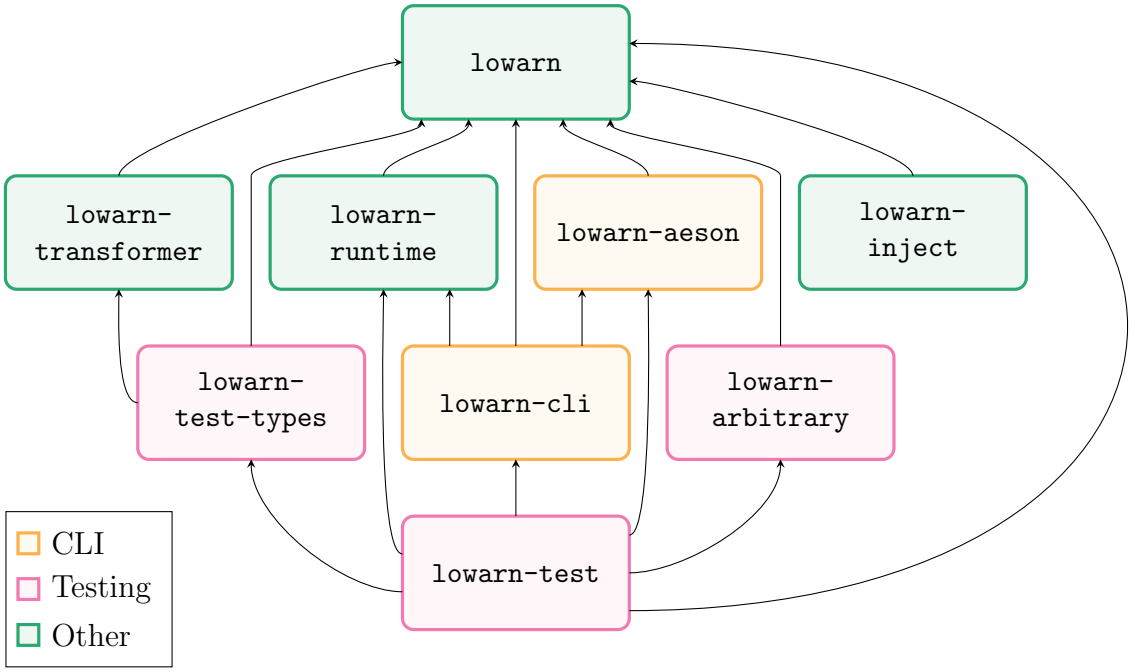


Figure 3.2: A dependency graph for the main repository.

Splitting the code into several packages makes it more modular and allows programmers to avoid incurring unwanted dependencies. The only two packages that are needed to achieve DSU are the core library (`lowarn`) and the runtime library. Heavier packages, such as the CLI library, do not need to be packaged with programs, making their binaries smaller and reducing the risk of dependency conflicts.

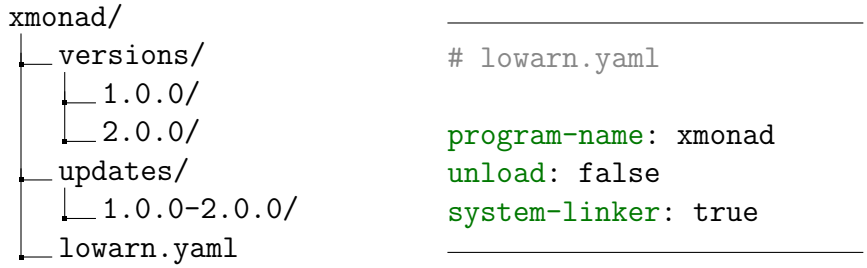
The core library is used by every other package and provides types used to represent versions and updates throughout the code. First, it provides types for program names, version numbers, version IDs and update IDs. These types are designed to avoid data redundancy and illegal values. Second, it includes types that need to be shared between the runtime and packages for versions and updates. These types, along with some lightweight metaprogramming facilities included with the core library, are described in the Section 3.3.

3.2 Command Line Interface

In this section, we describe our application's CLI. Its main purpose is to automatically run programs based on directory structure and configuration files. It also provides a suite

of tools that we use to easily retrofit many versions of existing programs by transferring changes as patch files.

Like Cabal and Stack, our CLI uses the concept of projects to group packages, as shown in Figure 3.3. A Lowarn project is a directory containing versions and updates in certain subdirectories as well as a YAML configuration file. Each version and update relates to a program specified in the configuration file, parsed with the Aeson serialisation library[1]. The configuration file also controls options of the DSU runtime described in Section 3.3. We search for the configuration file in the parent directories of the current working directory if it is not given as an argument.



(a) Directory structure. (b) Corresponding `lowarn.yaml`.

Figure 3.3: A simple Lowarn project.

Our automatic DSU runtime detects versions and updates using a directed graph data structure, where vertices are versions and edges are updates between versions. An example version graph is shown in Figure 3.4. We generate version graphs by searching the directory tree, checking that directories have Cabal configuration files and names that match certain parser combinators. The directory names resemble version numbers while the configuration file names resemble version and update IDs. The parser combinators for these types are tested with *property testing*, where we generate many valid values of a type and check that certain properties hold.

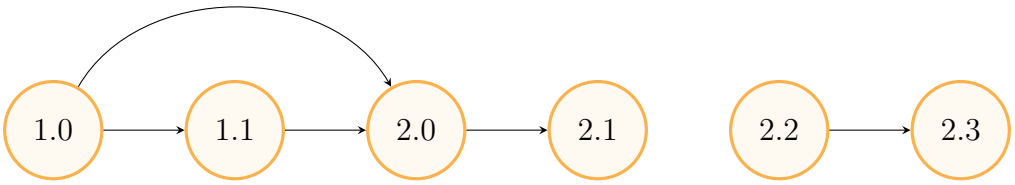


Figure 3.4: An example version graph.

The executable of our CLI takes many arguments and makes use of subcommands. It supports command-line completion and has a comprehensive help command. We achieve this through the `optparse-applicative` library[2], composing together argument parsers.

3.2.1 Retrofitting programs

The CLI can be used to retrofit existing program. By adding a Git repository to the configuration file, we can map increasing integer version numbers to commits. This requires us to find an arbitrary path through the commits of a repository, as they form a DAG.

[1] `https://github.com/haskell/aeson`. Retrieved April 24, 2023.
[2] `https://github.com/pcapriotti/optparse-applicative`. Retrieved April 24, 2023.

We create a canonical path through the commit history by taking the first parent of each commit from the head of a repository's main branch. Our CLI can clone a Git repository and use it to create this sequence of commits as a file that we can load with our *commit map* data structure. This is a string of bytes where every $2n$ bytes corresponds to the first n characters of a hexadecimal commit ID.

When retrofitting programs with the CLI, a version directory contains three subdirectories, each containing variants of the source. The source subdirectory is simply taken from the repository. The simplified subdirectory removes all unnecessary files and renames the package to be in the format required by the runtime. The retrofitted subdirectory modifies the simplified directory to add support for Lowarn DSU. The CLI creates patch files for simplifying and retrofitting that can be reasoned about and transferred across versions. We then ignore the subdirectories, as the patch files and directory structure make them redundant.

An example of a retrofitted program's project is given in Figure 3.5.

```
xmonad/
├── versions/
│   ├── 1.0.0/
│   │   ├── source/
│   │   ├── simplified/
│   │   ├── retrofitted/
│   │   ├── simplify.patch
│   │   └── retrofit.patch
│   └── 2.0.0/
│       └── ...
├── updates/
│   └── 1.0.0-2.0.0
└── lowarn.yaml
```

(a) Directory structure.

```yaml
# lowarn.yaml

program-name: xmonad
unload: false
system-linker: true
version-package-subdirectory: retrofitted

retrofit:
  git: https://xmonad.org/xmonad.git
  branch: master
```

(b) Corresponding `lowarn.yaml`.

Figure 3.5: A Lowarn project for a retrofitted program.

To create and apply the patch files, we make use of Git. However, recall that the version subdirectories are ignored by Git. Therefore, we cannot use Git's indexing. Despite this, we would like to make use of `.gitignore` files inside the subdirectories. Generating and patching therefore involves copying subdirectories to temporary directories, with ripgrep[3] being used to ignore files without using Git. Our integration tests ensure that these patches are generated correctly.

3.3 DSU Runtime

The most important feature of a DSU system is the ability to load code at runtime. We achieve this with a *DSU runtime*. This is a system that loads versions and updates of programs and threads state through them. It is also responsible for listening for and signalling when updates are available to perform.

[3] `https://github.com/BurntSushi/ripgrep`. Retrieved April 24, 2023.

3.3.1 Structuring programs

To achieve our DSU system's requirements, the runtime must be able to interact with the versions and updates. We have established that it is practical to represent versions and updates as packages. We will now discuss how we expose an interface for interacting with these packages.

As in Rubah and Kitsune, we would like to run a program as usual for the first version of a program and subsequently skip some control flow while running the following versions. Our approach is to let the programmer define an *entry point* function that can enter the program at programmer-defined points. We contain this function in a type called `EntryPoint a`, where `a` is the version's state type. The function takes runtime data, allowing it to access any previous state and decide the control flow to take. The runtime data can also be used to determine when an update is available. This indicates that the entry point function should return when possible, returning the most recent version of the state such that it can be transformed and transferred.

We represent state transformation with our `Transformer a b` type, a simple wrapper for a function of the form `a -> IO (Maybe b)`. This function can be seen as attempting to convert a value from type `a` to `b`. The `IO` part of the type signature allows side effects, while the `Maybe` part allows the function to fail. For simple functions, we provide the programmer with instances of type classes for function-like data, such as `Category` and `Arrow`. We also provide instances of type classes for computations that can fail, such as `Alternative`. This lets the programmer define a transformer that sequentially tries many potentially-failing transformers until one succeeds.

Initially, it made sense to have an update package only expose a value of `Transformer a b`. We now instead expose a value of the `Update a b` type, composed of a `Transformer a b` and a `EntryPoint b`. This makes the update package larger, but we no longer need to separately load a new version package when we perform an update, making loading faster. This type also introduces a degree of type safety, as the result type of the transformer and the state type of the entry point are required to be the same.

GHC provides multiple options for exposing a version package's entry point and an update package's update. Initially, we enforced that these were exposed as a top-level values with specific names in specific modules. These can be loaded by using a specific action in the GHC API. As explained in Section 3.3.3, we now instead expose these values using the C FFI.

Exposing a value with the C FFI can be verbose and complex, as we must expose pointers. Furthermore, we need to include version numbers in the export name to avoid conflicts in the symbol table. It is easy to forget to change these version numbers, as they are not statically checked. To solve both of these issues, we provide *Template Haskell* code in our core library that automatically creates the FFI exports given a top level value. Template Haskell[4] is a GHC extension that allows compile-time metaprogramming. Our Template Haskell code performs the unique task of acquiring and parsing the package name to find the correct program name and version numbers. One issue that we face with this approach is that IDEs that use Haskell Language Server[5] may raise errors for this code. This is because our code uses Template Haskell in a non-standard way that the language server does not expect. To keep IDEs usable, we detect when an IDE is attempting to run our Template Haskell code for its own analysis and generate no code, instead showing a harmless warning.

[5] `https://github.com/haskell/haskell-language-server`. Retrieved May 12, 2023.

3.3.2 Runtime monad

Our runtime library consists of two modules, each providing a *monadic type*. Monads are a key part of Haskell that can represent various useful concepts, as shown by Wadler [23]. We can make a type monadic by providing an instance of the `Monad` type class, providing implementations of two methods that follow certain laws. We have already seen the `IO` and `Maybe` monads in the `Transformer a b` type. We define monads to implement embedded domain specific languages (eDSLs) with actions that can be sequenced.

The runtime module contains the monadic `Runtime a` type, representing a DSU runtime that results in a value of type `a`. Its actions are shown in Table 3.1. It is mostly a wrapper for the linker monad, introduced in Section 3.3.3, but also handles runtime-specific behaviour, such as *update signal listening*.

Action	Description
`loadVersion versionId mPreviousState`	Loads a given version, optionally taking the state transformed from a previous version, giving the final state of the version.
`loadUpdate updateId previousState`	Loads a given update, taking the state of a previous version and giving the final state of the new version.
`updateRuntimePackageDatabase`	Updates the package database.

Table 3.1: Actions of the runtime monad.

Like in Kitsune and Rubah, the main way of triggering an update in Lowarn involves sending a SIGUSR2 signal to the process. When we execute a runtime, we create a binary semaphore that we call an update signal register, along with a signal listener that attempts to fill the register in a new thread when a SIGUSR2 is received. When we check for an update, we check if the register is filled, emptying it if so such that new updates can be subsequently triggered.

We provide the update signal register to every action in a runtime using a *reader* monad transformer. Monad transformers allow existing monads to be extended with new behaviours in a mostly composable way and are currently the most performant and common method for composing arbitrary effects in Haskell. The reader monad transformer wraps the linker monad, allowing the update signal register to be accessed, but not replaced, in all actions in the runtime monad. We use the update signal register to form the value of type `RuntimeData a` that the runtime gives to program entry points.

3.3.3 Linker monad

It is common to use dynamic linking to perform DSU. However, dynamic linking in Haskell is not as simple as dynamic linking in a language like C, as Haskell does not have a stable ABI and has a complicated runtime model for facilitating lazy evaluation. To use dynamic linking without being restricted to C types or experiencing segmentation faults, we make use of the GHC API.

Our linker module implements a `Linker a` monadic type that we use for actions involving linking, shown in Table 3.2. This wraps `Ghc a`, a monad provided by the GHC API that can be used to perform many of its tasks. The linker monad must be able to

start a GHC session and load in packages, fetching the data they expose. It must also be able to search the GHC package database to find the packages it must load.

Action	Description
`load packageName entityName`	Loads a given package and gives an entity with the given name exposed with the C FFI.
`updatePackageDatabase`	Updates the package database.

Table 3.2: Actions of the linker monad.

In its original form, the linker monad performed loading by setting a variety of flags and invoking an action that loads a module and any packages it depends on. Implementing this iteration of the linker allowed an early prototype of the DSU system to be developed. However, this method's simplicity restricts its flexibility. The current version of the linker uses a greater number of simple actions found in the GHC API to recreate this behaviour while allowing extension. This approach requires us to use the C FFI for exporting entities, as GHC only provides the functionality for reliably determining mangled names of top-level Haskell values with the previous method.

Forgoing our original method makes it harder to load a package and its dependencies. Unlike before, we are searching for a package, rather than a module. We can search the package database provided by GHC for package information. This gives all the package's direct dependencies, but we need the transitive closure of the package's dependencies. We perform a depth-first search of the unit database to find this, using a set, implemented as a size-balanced binary tree, to track already-visited dependencies.

When using Cabal, the package database does not include user-defined packages. To solve this, we must instruct Cabal to create a package environment file while building. This is given to the linker as an environment variable and loaded with certain GHC API actions. We also find that there is no clear way to reload the package database to find new packages at runtime. The solution to this problem involves various hacks.

As described in Section 2.2.2, we can either load static object files with the GHCi linker or shared object files with the system linker. We store which linker we use as data incorporated into the linker monad with a reader monad transformer. The package information provided by GHCi gives us the location of various build directories. Our linker must search these directories for the relevant files for the type of linking we are doing, ensuring that we are general enough in our search to work with both Cabal and Stack build systems.

Whenever the linker needs to load a package and its dependencies, it only needs to load the packages that haven't already been loaded. We use a *state* monad transformer to keep track of the set of currently loaded packages. This acts like the reader monad transformer but allows the value it stores to be replaced.

A key advantage of this design is that we can attempt to add unloading. We can add an option to the reader monad transformer that turns this behaviour on and use actions that allow static object files to be unloaded. For GHC's garbage collector to actually unload object files that we have requested to be unloaded, there must be no references to their top-level Haskell values. We achieve this by turning on an option called `DontRetainCAFs`, changing how Haskell garbage collects data it calls *constant applicative forms*. Unfortunately, this option can lead to segmentation faults with the GHCi linker in modern versions of GHC. This does not occur with the system linker, but in this case we

cannot unload code. Informal testing has shown that our DSU system can unload code, eventually keeping memory usage constant, but the occasional segmentation faults make this impractical for real programs.

3.3.4 Testing

To test that our DSU system works, we use an example program that keeps track of a list of users that are followed on a social media platform. The first version represents users with usernames stored in a list. The second version represents a user as a combination of a nickname and a four-digit ID, stored in a 2-3 finger tree annotated with sizes. In the update, the ID is randomly generated. The final version combines the nickname and ID into one string and reverts to using a list to store users.

This program can be used as a demo and as a base for integration tests in our test suite. Our integration tests use "golden testing", where the test writes to a file that is compared with an existing file. We introduce a monadic type `Story a`, used to resemble sequences of actions on programs such as sending input, reading output, and sending update signals. When a story is executed, we fork a subprocess with a given DSU runtime and log all the inputs and outputs of the process to the file, timing out if necessary. We run some of these tests with our CLI to test that this part of our system works.

3.4 Runtime Data Injection

As described in Section 3.1, it is sometimes necessary for a program to access a value of type `RuntimeData a`. This is provided at the program's entry point, but may be needed at any point in the program. We would therefore like to be able to access it "globally". However, Haskell does not support semantically-safe global variables, as initialising a global value requires a side effect when Haskell is supposed to be pure. In this section, we use a GHC typechecker plugin to make runtime data available throughout a package while still maintaining some safety.

The runtime data injection library provides two type classes that allow programmers to store and take runtime data respectively. Instances of these type classes are automatically generated by our typechecker plugin. The generated methods access a top-level *runtime data variable* found in a given module in the program, shown in Listing 4. This type wraps a synchronising variable that may store a runtime data value. We must be careful with this top-level value, as it uses an unsafe trick that allows it to have side effects while existing outside the `IO` monad.

First, the typechecker plugin must locate the functions and types that it uses. At this stage, we also enforce rules that reduce the danger introduced by runtime data injection. In particular, we require that the plugin only works in version packages and that we only inject data in a version's entry point module.

When our plugin is invoked to solve wanted constraints, we locate all constraints for our library's type classes. Examples of constraints and how they can be introduced are shown in Table 3.3. To solve constraints for injecting data, we can simply generate a dictionary in Core form that writes to the program's runtime data variable. Solving constraints for accessing injected data is more complicated. We need the evidence dictionary to have a specific type that depends on the relevant method's usage. Table 3.3 shows how this type may include a type variable if we do not need the state. This type is proven to be

```haskell
module RuntimeDataVar_xmonad (runtimeDataVar) where

import GHC.IO (unsafePerformIO)
import Lowarn.Inject.RuntimeDataVar (RuntimeDataVar, newRuntimeDataVar)
import {-# SOURCE #-} Xmonad (State)
-- The SOURCE pragma allows us to have a circular module dependency.

{-# NOINLINE runtimeDataVar #-}
runtimeDataVar :: RuntimeDataVar State
runtimeDataVar = unsafePerformIO newRuntimeDataVar
```

Listing 4: An example runtime data variable module for a program.

Constraint to solve	**Example code**

`InjectRuntimeData State`	`entryPoint :: EntryPoint State`
	`entryPoint = EntryPoint $`
	`  \runtimeData -> do`
	`  injectRuntimeData runtimeData`
	`  eventLoop (lastState runtimeData)`

`InjectedRuntimeData a`	`eventLoop :: State -> IO State`
	`eventLoop state = do`
	`  runtimeData <- injectedRuntimeData`
	`  shouldStop <- isUpdateAvailable runtimeData`
	`  if shouldStop then return state else ...`

`InjectedRuntimeData State`	`printLastState :: IO ()`
	`printLastState = do`
	`  runtimeData <- injectedRuntimeData`
	`  let previousState = lastState runtimeData`
	`  print (previousState :: State)`

Table 3.3: Example wanted constraints for the runtime data injection plugin and how they are introduced.

equal to the type of our generated Core code using a Core coercion. We introduce a new hole-constraint pair corresponding to this desired coercion and return evidence including this hole, as well as this new wanted constraint for the constraint solver to solve.

One issue is that the runtime data variable module must be built before every module that uses the plugin. We guarantee this by importing this module, but not its contents, in every other module, using a *preprocessor*. This is an executable used by GHC to change Haskell source code in text form. Our preprocessor parses Haskell modules with parser combinators and chooses to import the required module and enable the plugin when possible. With a preprocessor, we are also able to automatically generate the code of runtime data variable module, reducing Listing 4 to two lines.

To test the runtime data injection library, we add it to one of the testing programs described in Section 3.3.4 and check that the integration tests continue to pass. Various documentation tests also verify the output of the preprocessor.

3.5 Type-Driven State Transformations

When we update from one version of a program to another, we must carry over the state. However, types defined in one version of a program will not be interchangeable with types defined in the same way in a following version of the program, as the modules that they are defined in are different. As a result, a state transformer for two state types may require many other simple transformers that simply map between representationally-equivalent types. Our transformers library, inspired by type-driven transformation tools for Kitsune and Rubah like xfgen, allows simple state transformations to be automatically derived while still allowing different types to have manually-defined transformers. We can derive transformers between the following types:

- A type and the exact same type;
- The type of a left-to-right traversable data structure, containing values of a type, and the same data structure with the inner type being transformed with a known transformer;
- An algebraic data type (ADT) and the result of transforming each inner type of the ADT with a known transformer and changing the constructor and field names;
- An ADT and the result of reordering the constructors and fields of the ADT and transforming each inner type with a known transformer;
- A type and another type that has the same data representation at runtime.

Our transformer library introduces the `Transformable a b` type class. This type class has one method, `transformer`, that gives a `Transformer a b` value and another method, `transform`, of the type `a -> IO (Maybe b)`. Only one of these methods needs to be provided to declare an instance. The power of the transformer library comes from `Transformable a b` being automatically generated when types `a` and `b` are similar. For example, two ADTs that contain the same types in the same places can have an automatic transformable instance if every corresponding pair of contained types also has a transformable instance. As demonstrated in Listing 5, this means that we typically only need to define transformers for types that have actually changed.

As mentioned in Section 2.3.5, we use the `generics-sop` library for most transformer generation. However, this library requires types to have instances of the `Generic` type class. This is not possible for *unboxed* types, types that include existentially-qualified type variables (type variables that only exist on the right-hand side of the type), or GADTs (datatypes where the constructor types can be given explicitly). However, we are still able to provide some support for working with these types. First, we always have instances of `Transformable a a`, the identity transformer. Furthermore, we have instances of `Transformable (t a) (t b)` when we have an instance `Transformable a b`, where `t` is the type constructor of a left-to-right traversable data structure, such as a list. Finally, we can define a transformer for any two types that are *representationally equivalent* (have the same data representation at runtime) using a method introduced by Breitner et al. [24] called *safe coercions*. This method is not used in automatic transformer generation, as it may produce transformers between unrelated types, so we instead provide a function for it that can be explicitly used.

```haskell
-- PreviousVersion.hs

data Coord = Coord { x :: Int, y :: Int }
data Colour = Colour { r :: Int, g :: Int, b :: Int }
data Circle = Circle { pos :: Coord, radius :: Int, colour :: Colour }
newtype Circles = Circles { circles :: [Circle] }

-- NextVersion.hs

data Coord = Coord { x :: Int, y :: Int }
data Colour = Colour { r :: Int, g :: Int, b :: Int, alpha :: Int }
data Circle = Circle { pos :: Coord, radius :: Int, colour :: Colour }
newtype Circles = Circles { circles :: [Circle] }

-- Update.hs

instance Transformable PreviousVersion.Colour NextVersion.Colour where
  transform :: PreviousVersion.Colour -> IO (Maybe NextVersion.Colour)
  transform (PreviousVersion.Colour r g b) =
    return $ Just $ NextVersion.Colour r g b 255

-- Assume we derive type classes from generics-sop for the above types.
-- We simply use transformer, a method of Transformable a b.
circlesTransformer ::
  Transformer PreviousVersion.Circles NextVersion.Circles
circlesTransformer = transformer
```

Listing 5: The `Equals` type family.

We define three functions that transform data generically. The generic transformer only looks at the structure of the two types. The other two functions additionally makes use of metadata. The generic renaming transformer acts the same as the generic transformer but only type-checks when datatype, constructor, or field names are the same or have been explicitly renamed, while the generic reordering transformer reorders constructors and fields using their names and then transforms.

Transformable instances for generic transformations necessarily have the right-hand side `Transformable a b`, where `a` and `b` match all types of kind `Type`, with the constraints that we use to infer the transformations being found on the left-hand side. We are not able to define multiple instances with the same right-hand side, so we need to choose one of the three methods for generic transformations to use automatically. We use the renaming method automatically, as this is the safest and most flexible method, but the other methods can still be used explicitly with provided functions.

Instance right-hand sides cause further issues when they differ but overlap. When GHC is determining the instance to use for a method of a type class, it only checks the type variables of the right-hand side of the instance declaration. If two or more instances match, we have ambiguity and the program will not type-check. The right-hand side of the generic transformer matches all pairs of datatypes, preventing the definition of manual instances. We also encounter a similar issue when finding automatic instances for types

such as `Transformable [a] [a]`, where the right-hand sides of the identity, traversable, and generic instances all simultaneously match.

To allow instances to safely overlap, GHC provides a pragma, `{-# OVERLAPPABLE #-}`, that stops an instance being applicable when there is an overlap. Giving each automatic instance this pragma facilitates the definition of manual instances, but we still have ambiguity when we must choose between only automatic instances. We solve this by having only one overlappable instance `Transformable a b` where the constraints first form a type equality between `Equals a b`, from Section 2.3.3, and a new type variable `p`, then use an instance of an internal type class `Transformable' p a b`. The identity instance is given as `Transformable' 'True a a` and the traversable and generic instances as `Transformable' 'False (t a) (t b)` and `Transformable' 'False a b` respectively, with the latter being overlappable such that the former is prioritised.

3.5.1 Generic transformers

In this subsection, we use types introduced in Section 2.3.5 to implement simple generic transformers for ADTs like those shown in Listing 6.

```
instance Transformable Coord NewCoord
instance Transformable Colour NewColour
instance Transformable Rect NewRect

data CircleA = CircleA
  { circleAPos :: Coord,
    circleARadius :: Int,
    circleBColour :: Colour
  }

data CircleB = CircleB
  { circleBPos :: NewCoord,
    circleBRadius :: Int,
    circleBColour :: NewColour
  }

data ShapeA = ShapeACircle CircleA | ShapeARect Rect
data ShapeB = ShapeACircle CircleB | ShapeARect NewRect
```

Listing 6: Two pairs of ADTs that we can generate generic transformers between, given the instances at the top.

The constraints of our generic transformer function ensure that the sum of products representations of the two types have the same shape and that each pair produced by zipping the representations together is composed of two types where we have a transformable instance. We would like to use the latter constraint to run a function that applies each transformer to the inner types. We can use the `htrans` function provided by `generics-sop` to do this. However, recall that transformers return values of type `IO (Maybe a)`. We do not want to end up with a type `SOP I xss` where every type in `xss` is of the form `IO (Maybe a)`. This is where the power of the SOP type's functor

become apparent. We note that `IO` and `Maybe` are both functors and combine them into one functor using the `(:.:)` (functor composition) type. Using `htrans`, we produce an SOP with the type `SOP (IO :.: Maybe) yss`, where `yss` contains the inner types of the result type. We can then apply *sequencing* functions that produce a type of the form `IO (Maybe (SOP I yss))`, as required.

3.5.2 Generic renaming transformers

The generic transformer described in Section 3.5.1 will transform similar types without considering datatype metadata. Consider the pair of types shown in Listing 7.

```
data LengthInCentimeters = LengthInCentimeters
  { LengthInCentimeters :: Int
  }

data LengthInMillimeters = LengthInMillimeters
  { LengthInMillimeters :: Int
  }
```

Listing 7: A pair of ADTs that we can generate a generic transformer between, but not a generic renaming transformer between.

While we can clearly generate a generic transformer between `LengthInCentimeters` and `LengthInMillimeters`, the different datatype, constructor, and field names clearly indicate that this transformer would not be correct. In particular, we need to multiply the integer in the former type by 10 to get the integer in the latter type. The generic renaming transformer, used for automatic instances of `Transformable a b`, will not have an instance for this pair of types, preventing an incorrect transformation. If the name changes are superficial, like they are in Listing 6, we can provide instances of *alias* type classes to make the generic renaming transformer type-check, as shown in Listing 8.

```
instance DatatypeNameAlias "CircleA" "CircleB"
instance ConstructorNameAlias "CircleA" "CircleB"
instance FieldNameAlias "circleAPos" "circleBPos"
instance FieldNameAlias "circleARadius" "circleBRadius"
instance FieldNameAlias "circleAColour" "circleBColour"

instance DatatypeNameAlias "ShapeA" "ShapeB"
instance ConstructorNameAlias "ShapeACircle" "ShapeBCircle"
instance ConstructorNameAlias "ShapeARect" "ShapeBRect"
```

Listing 8: Aliases required to generate generic renaming transformers between pairs of datatypes in Listing 6.

The generic renaming transformer uses the same function as the generic transformer but adds additional constraints based on metadata, provided by the `HasDatatypeInfo` type class. This allows us to get type-level data that we can manipulate with our own

type families to produce type-level strings for datatype, constructor, and field names. We use the zipping method mentioned in Section 3.5.1 to compare pairs of names and check for alias instances.

We automatically generate identity instances for the alias type classes. These must be overlappable to allow manual instances. This is the reason why we do not represent aliases as open type families, which do not support overlapping instances.

3.5.3 Generic reordering transformers

We implement the generation of transformers that reorder a datatype's fields and constructors, as demonstrated in Appendix B.

This is complicated, as we can no longer rely on functions such as `htrans` that do not expect the structure of types to change. We would like to pair each sum and product type with a type-level list of names and rearrange this type-level list to match the names of the type that we are transforming to, while simultaneously rearranging the sum and product types in the same way. We must pair each constructor of the sum type with a type-level list of names paired with the type-level list used for the inner product types. We introduce some special type-level data to represent this.

To perform this procedure, described with Prolog logic rules in Appendix B, we must perform logical inference of type variables. We use the associated type family method to achieve this, as explained in Section 2.3.3.

Reordering is incompatible with renaming. Our algorithm requires that we can identify if two type-levels strings are the same or not the same. This is possible with type-level equalities. However, it is not possible to identify whether an instance of an alias type class does not exist under Haskell's open-world assumption. We can use typechecker plugins to break this rule, but this would break other assumptions we make, making our transformers no longer type-safe.

3.5.4 Testing

As part of testing our transformer library, we need to confirm that certain pairs of types, such as those where a renaming has occurred without instances of alias type classes being given, do not have instances of `Transformable a b`. To test that code does not type-check, we use `hint`[6], a library that uses GHCi to type-check and run Haskell code. Our tests use a wide range of datatypes, contained in their own package as required by `hint`.

[6]https://github.com/haskell-hint/hint. Retrieved April 24, 2022.

3.6 Repository Overview

Our code is found in five Git repositories, shown in Table 3.4. The `lowarn` repository contains the packages composing the Lowarn DSU system, along with tests and examples. The `lowarn-evaluate` repository contains programs and scripts used in evaluation. The three other repositories contain patches and updates for versions of real-world Haskell programs.

Directory tree		Files	Lines[7]
/		**131**	**9,310**
`lowarn/`		**114**	**8,362**
`core/`	Code used by all packages.	8	1,059
`runtime/`	DSU runtime and linker.	3	418
`transformer/`	Type-driven transformation library.	3	883
`inject/`	Runtime data injection library.	7	649
`cli/`	CLI.	12	1,651
`aeson/`	JSON/YAML conversions for core types.	5	112
`test/`	Tests for all the above packages.	20	2,245
`arbitrary/`	Property testing generators for core types.	5	142
`test-types/`	Types for testing transformers.	9	426
`examples/`	Example programs for testing.	**41**	**767**
`following/`	Uses all DSU features.	20	363
`manual-following/`	Uses basic DSU features.	17	336
`custom-ffi/`	Uses the C FFI.	4	68
`coverage.sh`	Script that calculates coverage.	—	10
`lowarn-evaluate/`		**17**	**948**
`retrofit/`	Scripts for retrofitting.	5	81
`collect-csv/`	Result file consolidation program.	2	173
`xmonad-benchmark/`	`xmonad` benchmark.	2	158
`smp-server-benchmark/`	`smp-server` benchmark.	2	204
`plot/`	Scripts for plotting.	6	332
`lowarn-xmonad/`		—	—[8]
`lowarn-smp-server/`		—	—
`lowarn-hnes/`		—	—

Table 3.4: Overview of repositories.

3.7 Summary

We developed a framework for performing DSU on real-world long-running programs. We implemented a DSU runtime for this framework as well as several libraries that increase the system's practicality. In particular, we introduce a safe and extensible library for type-driven state transformations, a compiler plugin for easy runtime data injection, and a CLI that improves developer experience.

[7] These file and line counts are computed with `cloc` for Haskell, C, `sh`, and Python files. Configuration files for build systems and CI are not counted.

[8] The file and line counts for existing programs are omitted due to update code and patches being repeated many times.

4 Evaluation

In this chapter, we evaluate the project's success against the following success criterion, given in the proposal found in Appendix C:

A successful DSU system for long-running programs should be able to perform many updates on real-world programs.

In Section 4.1, we show that this criterion is met by performing all but two of of 237 updates for three long-running Haskell programs. In Section 4.2, we introduce benchmarks for two of these programs such that we can further evaluate our DSU system's impact on performance in Section 4.3 and Section 4.4.

4.1 Updates performed

It is not possible to verify that Lowarn can perform updates on every real-world program. Therefore, like in most existing work on DSU, we evaluate our DSU system against a varied sample of open source projects. By using well-established and varied long-running Haskell programs, we ensure that this sample is representative. Furthermore, we perform many updates on each program such that our samples of updates are representative.

4.1.1 `xmonad`

The first long-running Haskell program that we evaluate our DSU system against is `xmonad`, the executable of a tiling window manager introduced by Stewart and Sjanssen [25] for the X window system. This program has been developed and maintained since 2007. With the methodology for reasoning about Git commits described in Section 3.2, `xmonad`'s Git history contains around 1464 successive commits, at the time of writing. However, `xmonad` previously used the Darcs version control system. The conversion from Darcs to Git has failed to preserve the meaning of the first 1141 commits. Furthermore, only 97 of the commits from commit 1142 to commit 1464 actually change the code of the window manager. Some earlier commits also required that we backport some simple changes to support modern versions of GHC.

We were able to successfully declare and perform updates for each of the 97 versions of `xmonad` in our sample.

4.1.2 `smp-server`

The second long-running Haskell program that we evaluate our DSU system against is `smp-server`[1], a message broker for the SimpleX Chat messaging platform[2]. It is inspired by Redis, a long-running program previously used to evaluate Kitsune. `smp-server` is found in a Git repository that contains about 356 successive commits, at the time of

[1] `https://github.com/simplex-chat/simplexmq`. Retrieved May 09, 2023.
[2] `https://simplex.chat/`. Retrieved May 09, 2023.

writing. We test the 257th to 386th commits, with 100 of these commits changing the code. This set of commits spans October 2022 to May 2023.

We were able to successfully declare and perform updates for 99 of the 100 commits. The one commit that could not be performed added the number of notifications sent so far as a statistic. This data cannot be checked retrospectively, so the update is not possible.

4.1.3 `hnes`

The third long-running Haskell program that we evaluate our DSU system against is `hnes`[3], an emulator for the NES video game console. It has been in development since 2017 and has around 210 successive commits, at the time of writing. We test 40 commits between the 147th to 210th commits, with versions of emulator before this being unstable.

We were able to successfully declare and perform updates for 39 of the 40 commits. The one impossible updates involves finding the memory layout of some data that only exists at initialisation.

As `hnes` does not use meaningful releases, we show summaries for every 12 commits.

4.1.4 Results

We were able to perform all but two of the updates in our samples. In Table 4.1, we show the number of lines needed to be added or modified for versions at major milestones. We find that these numbers remain relatively constant and do not depend on the size of the programs, except for the lines of new imports, pragmas, and type classes. These lines depend on the number of modules and types in the program. The update package's size also varies based on the program.

[3] `https://github.com/dbousamra/hnes`. Retrieved May 12, 2023.

Program version (commit/ release)	**# relevant[4] lines in source**	**# lines in new files**	**# lines of minor changes[5]**	**# other lines modified**	**# lines in update file**
xmonad					
1,147/0.11	1,430	28	37	30	114
1,163/0.12	1,453	28	37	30	117
1,223/0.13	1,661	28	38	39	119
1,234/0.14	1,688	28	38	39	119
1,271/0.15	1,734	28	38	39	119
1,326/0.16	1,724	29	39	37	123
1,363/0.17	1,823	29	42	42	123
smp-server					
268/3.3	13,725	60	59	13	223
289/3.4	14,400	60	59	13	223
300/4.0	14,867	70	60	13	224
308/4.1	14,958	71	60	13	260
317/4.2	15,117	71	60	13	224
327/4.3	15,188	71	60	13	224
338/4.4	15,628	71	60	13	224
356/5.0	15,936	71	64	13	224
hnes					
152/0.1	1,972	2	31	45	38
165/0.1	2,146	2	45	50	90
180/0.1	2,074	2	44	50	90
207/0.1	2,083	2	44	52	90

Table 4.1: Lines of code modified to retrofit milestone versions of programs.

In the case of `hnes`, lots of logic exists in the executable rather than the library, so we turn this executable into the entry point instead of making a new entry point module. As a result, this program has fewer lines of code in new files and more lines of code modified in other ways.

4.2 Benchmarks

To evaluate the effects of our DSU system on speed and memory usage, we need benchmarks for our real-world programs. However, there are no any existing benchmarks for these programs that we can use, so we introduce new ones in this section.

In the X Window System, `xmonad` is an X client that talks to an X server. We can therefore benchmark it by stopping the process, queueing 6000 events with the XTEST protocol, resuming the process, and waiting for it to handle all the events. This is timed by listening to certain X events until we receive no events related to window manager activity for a certain amount of time. We stop the process and queue the events such that we do not accidentally measure how long it takes to send 6000 events.

[4]Relevant lines exclude unused files and include backported compatibility fixes.

[5]Minor changes consist of new imports, pragmas, and type class instances.

To benchmark `smp-server`, we create 50 pairs of clients. In each pair, one client sends 100 messages to the other. We time how long it takes for all the messages to be sent and received.

Our experiments involve timing how long benchmarks take while constantly measuring the memory usage, with and without the DSU system. We repeat each experiment 10 times to improve accuracy. We also introduce a delay between each experiment such that the system can stabilise and to reduce the risk of external background events affecting the data.

Measurements were collected on a PC with the following specification:

Processor. Intel Core i7-10510U 1.80 GHz.

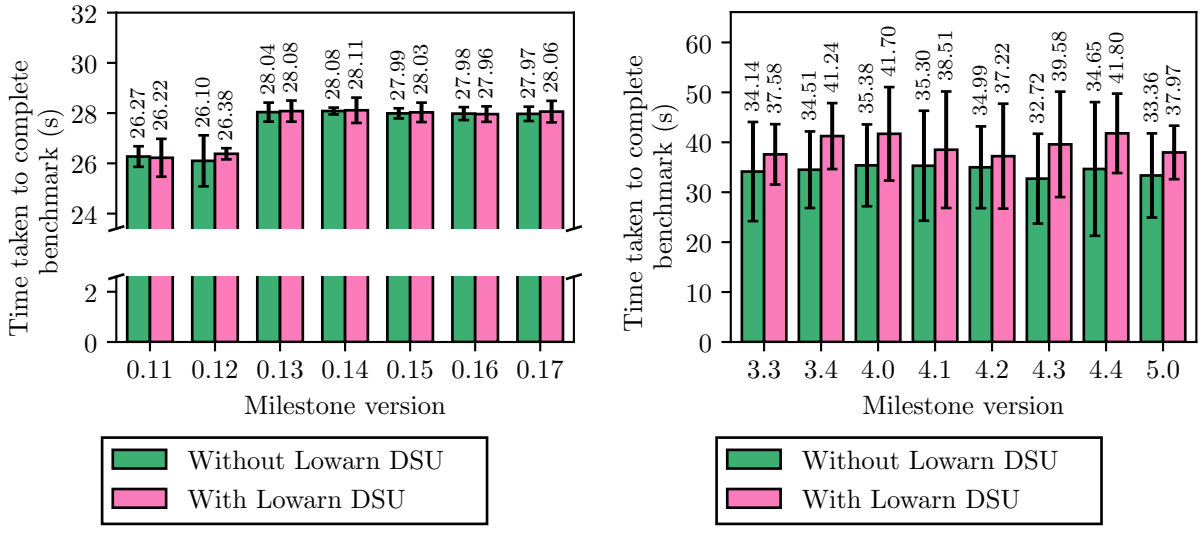
Memory. 8 GiB 2666 MHz DDR4 RAM.

Storage. 500 GB M.2 NVMe SSD.

OS. Ubuntu 20.04.6 LTS.

4.3 Impact on speed

In Figure 4.1, we show the time taken to perform our benchmarks.



(a) `xmonad` (b) `smp-server`

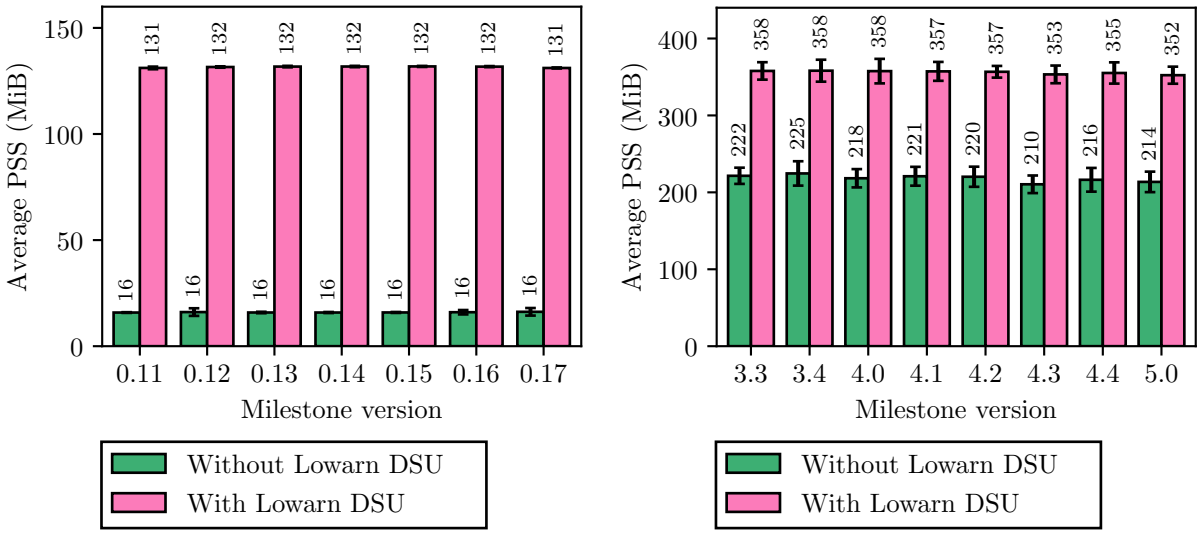
Figure 4.1: Times taken to perform our benchmarks, with and without Lowarn DSU. Error bars represent $\pm 2\sigma$.

The graphs show how running `xmonad` with Lowarn has no effect on performance in terms of speed, but they do show an overhead for `smp-server`. This is because `smp-server` has a memory-bound benchmark, with 100 clients being handled concurrently. As described in Section 4.4, Lowarn does have a memory overhead.

4.4 Impact on memory

To measure memory usage, we use `smem`[6] to find the proportional set size (PSS) over time. This measures the main memory usage of a process by adding its private memory to a proportion of its shared memory, giving a more accurate measure than measures such as resident set size.

In Figure 4.2, we show the average PSS used while performing our benchmarks.



(a) `xmonad` (b) `smp-server`

Figure 4.2: Average Proportional Set Size (PSS) while performing our benchmarks, with and without Lowarn DSU. Error bars represent $\pm 2\sigma$.

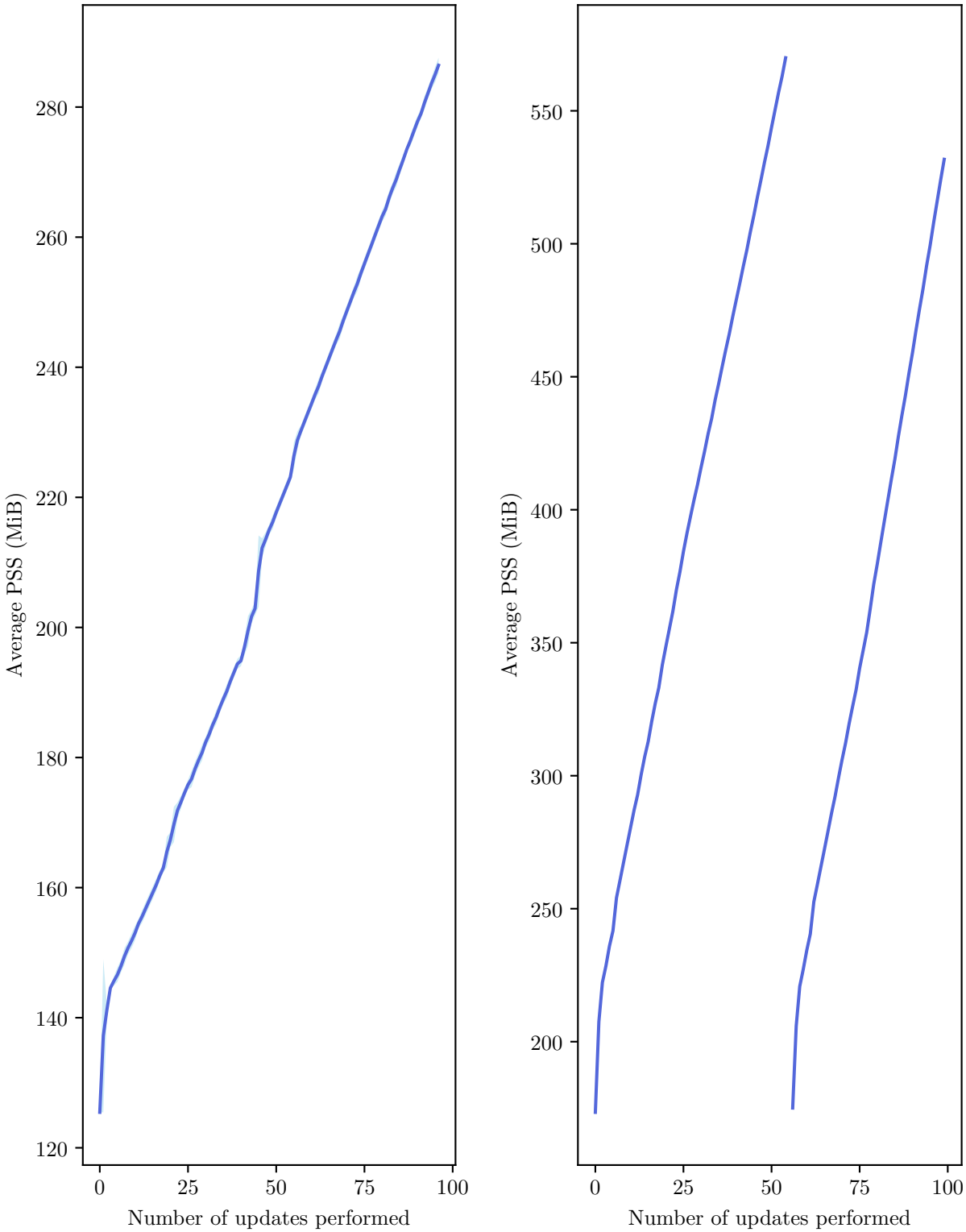
The graphs show how running a program with Lowarn significantly increases the memory usage by over 100 MiB. This can be attributed to additional dependencies and the overhead of our DSU runtime system.

In Figure 4.3, we show the memory usage of idle processes of `xmonad` and `smp-server` running with Lowarn as we perform an update every five seconds. As we were unable to fully implement unloading in Section 3.3, the memory usage increases linearly over time. There is a large jump when the first update is loaded, as we must load new dependencies that remain loaded beyond this point.

4.5 Summary

Our DSU system meets its success criteria, performing all but two of the updates in our samples. Furthermore, our results suggest that there is not too much of an impact on speed. We also find that using Lowarn requires more memory and confirm that memory usage increases linearly as we load updates.

[6] `https://www.selenic.com/smem/`. Retrieved April 24, 2023.



(a) `xmonad` (b) `smp-server`

Figure 4.3: Average Proportional Set Size (PSS) of `xmonad` and `smp-server` retrofitted with Lowarn DSU, after successive updates are performed. The shaded regions represent $\pm 2\sigma$. The discontinuity in the second graph represents the process restarting due to an update not being possible.

5 Conclusions

In this project, we aimed to create a new DSU system for Haskell that allows existing programs to be easily retrofitted. We succeeded in this goal, introducing a system that allowed us to update a number of well-established Haskell programs by making small numbers of changes.

Key innovations in our project include composable DSU runtime that loads packages rather than modules (Section 3.3), a type-safe and extensible mechanism for automatically deriving state transformers (Section 3.5), a compiler plugin for injecting runtime data (Section 3.4), and a CLI that allows the developer to easily run updates and streamlines retrofitting many versions of the same program (Section 3.2). Throughout these parts of our DSU system, we make full use of a variety of advanced features of Haskell to ensure safety and ease-of-use.

5.1 Future work

Unfortunately, bugs in modern versions of GHC prevented us from adding package unloading to our DSU system, resulting in linearly increasing memory usage as we load updates. In the future, we would like to fix these bugs such that we can take full advantage of our system's design.

In our evaluation, we informally verified the correctness of updates by running the code. Our work could benefit from making use of a framework for running tests on programs that have undergone and are undergoing updates, introduced by Pina and Hicks [26].

Finally, we could consider adding the label system extension from Section 2.4. Although it was not necessary to update any of the programs we evaluated our DSU system with, it could be useful in others.

5.2 Lessons learnt

The timetable found in the project proposal was overly optimistic, as certain tasks, such as implementing the DSU runtime, took longer than expected. Fortunately, it included sufficient slack time to allow me to catch up.

At certain stages of development, I attempted to solve very difficult problems, including code unloading and reordering transformers. While many of these problems were eventually solved, I believe that these difficult problems should have been left for later on in order to prioritise more important features, such as the CLI.

Bibliography

[1] Pablo Tesone et al. "Dynamic software update from development to production". In: *The Journal of Object Technology* 17.1 (Nov. 2018), pp. 1–36. DOI: https://doi.org/10.5381/jot.2018.17.1.a2.

[2] Alex McLean. "Making programming languages to dance to: live coding with Tidal". In: *Proceedings of the 2nd ACM SIGPLAN International Workshop on Functional Art, Music, Modeling & Design*. FARM 2014. Sept. 2014, pp. 63–70. DOI: https://doi.org/10.1145/2633638.2633647.

[3] Manuel Bärenz. "The essence of live coding: change the program, keep the state!" In: *Proceedings of the 7th ACM SIGPLAN International Workshop on Reactive and Event-Based Languages and Systems*. REBLS 2020. Nov. 2020, pp. 2–14. DOI: https://doi.org/10.1145/3427763.3428312.

[4] Pankaj More. "Hot Code Reloading in Cloud Haskell". PhD thesis. Indian Institute Of Technology Kanpur, 2014.

[5] Don Stewart. "Dynamic extension of typed functional languages". PhD thesis. UNSW Sydney, 2010. DOI: https://doi.org/10.26190/unsworks/23404.

[6] John Peterson, Paul Hudak, and Gary Shu Ling. *Principled dynamic code improvement*. Research Report YALEU/DCS/RR-1135. Department of Computer Science, Yale University, July 1997.

[7] Babiker Hussien Ahmed et al. "Dynamic software updating: a systematic mapping study". In: *IET Software* 14.5 (Sept. 2020), pp. 468–481. DOI: https://doi.org/10.1049/iet-sen.2019.0201.

[8] Christopher M. Hayden et al. "Evaluating dynamic software update safety using efficient systematic testing". In: *IEEE Transactions on Software Engineering* 38.6 (Dec. 2012), pp. 1340–1354. DOI: https://doi.org/10.1109/TSE.2011.101.

[9] Gautam Altekar et al. "OPUS: online patches and updates for security". In: *Proceedings of the 14th Conference on USENIX Security Symposium*. SSYM 2005. July 2005, p. 19.

[10] Iulian Neamtiu et al. "Practical dynamic software updating for C". In: *Proceedings of the ACM Conference on Programming Language Design and Implementation*. PLDI 2006. June 2006, pp. 72–83. DOI: https://doi.org/10.1145/1133981.1133991.

[11] Christopher M. Hayden et al. "Kitsune: efficient, general-purpose dynamic software updating for C". In: *Proceedings of the ACM Conference on Object-Oriented Programming Languages, Systems, and Applications*. OOPSLA 2012. Oct. 2012, pp. 249–264. DOI: https://doi.org/10.1145/2384616.2384635.

[12] Luís Pina, Luís Veiga, and Michael Hicks. "Rubah: DSU for Java on a stock JVM". In: *Proceedings of the ACM Conference on Object-Oriented Programming Languages, Systems, and Applications*. OOPSLA 2014. Oct. 2014, pp. 103–119. DOI: https://doi.org/10.1145/2660193.2660220.

[13] Simon Marlow et al. *Haskell 2010 language report*. Tech. rep. Available at https://www.haskell.org/definition/haskell2010.pdf. July 2010.

[14] Simon Peyton Jones et al. “A reflection on types”. In: *A List of Successes That Can Change the World: Essays Dedicated to Philip Wadler on the Occasion of His 60th Birthday*. Ed. by Sam Lindley et al. Springer International Publishing, 2016, pp. 292–317. DOI: https://doi.org/10.1007/978-3-319-30936-1_16.

[15] Brent A. Yorgey et al. “Giving Haskell a promotion”. In: *Proceedings of the 8th ACM SIGPLAN Workshop on Types in Language Design and Implementation*. TLDI 2012. Jan. 2012, pp. 53–66. DOI: https://doi.org/10.1145/2103786.2103795.

[16] Stephanie Weirich, Justin Hsu, and Richard A Eisenberg. “System FC with explicit kind equality”. In: *ACM SIGPLAN Notices* 48.9 (Sept. 2013), pp. 275–286. DOI: https://doi.org/10.1145/2544174.2500599.

[17] Simon Peyton Jones, Mark Jones, and Erik Meijer. “Type classes: an exploration of the design space”. In: *Haskell workshop*. Jan. 1997. URL: https://www.microsoft.com/en-us/research/publication/type-classes-an-exploration-of-the-design-space/.

[18] Simon Peyton Jones. *Type inference as constraint solving: how GHC's type inference engine actually works*. Zurihac keynote talk. June 2019. URL: https://www.microsoft.com/en-us/research/publication/type-inference-as-constraint-solving-how-ghcs-type-inference-engine-actually-works/.

[19] Robin Milner. “A theory of type polymorphism in programming”. In: *Journal of Computer and System Sciences* 17.3 (Dec. 1978), pp. 348–375. DOI: https://doi.org/10.1016/0022-0000(78)90014-4.

[20] José Pedro Magalhães et al. “A generic deriving mechanism for Haskell”. In: *Proceedings of the Third ACM Haskell Symposium on Haskell*. Haskell '10. Sept. 2010, pp. 37–48. DOI: https://doi.org/10.1145/1863523.1863529.

[21] Edsko de Vries and Andres Löh. “True sums of products”. In: *Proceedings of the 10th ACM SIGPLAN workshop on Generic programming*. WGP '14. Aug. 2014, pp. 83–94. DOI: https://doi.org/10.1145/2633628.2633634.

[22] Barry W. Boehm. “A spiral model of software development and enhancement”. In: *Computer* 21.5 (May 1988), pp. 61–72. DOI: https://doi.org/10.1109/2.59.

[23] Philip Wadler. “Monads for functional programming”. In: *Advanced Functional Programming: First International Spring School on Advanced Functional Programming Techniques*. May 1995, pp. 24–52. DOI: https://doi.org/10.5555/647698.734146.

[24] Joachim Breitner et al. “Safe zero-cost coercions for Haskell”. In: *Proceedings of the 19th ACM SIGPLAN international conference on Functional programming*. Aug. 2014, pp. 189–202. DOI: https://doi.org/10.1145/2692915.2628141.

[25] Don Stewart and Spencer Sjanssen. “Xmonad”. In: *Proceedings of the ACM SIGPLAN Workshop on Haskell Workshop*. Haskell '07. Sept. 2007, p. 119. DOI: https://doi.org/10.1145/1291201.1291218.

[26] Luís Pina and Michael Hicks. “Tedsuto: a general framework for testing dynamic software updates”. In: *2016 IEEE International Conference on Software Testing, Verification and Validation (ICST)*. Apr. 2016, pp. 278–287. DOI: https://doi.org/10.1109/ICST.2016.27.

A Summary of licences

Executable	Licence type	Count
`lowarn-cli`	BSD 3-Clause	82
	MIT	13
	BSD 2-Clause	4
	ISC	1
`lowarn-inject`	BSD 3-Clause	32
	MIT	2
`collect-csv`	BSD 3-Clause	80
	MIT	13
	BSD 2-Clause	4
	ISC	1
`xmonad-benchmark`	BSD 3-Clause	24
	BSD 2-Clause	1
`smp-server-benchmark`	BSD 3-Clause	119
	MIT	21
	BSD 2-Clause	5
	ISC	1
	AGPL-3.0-only	1
	Public domain	1

Table A.1: Licences associated with major executables and their transitive dependencies.

B Reordering logic

Consider the following types in two different modules:

```haskell
-- PreviousVersion.hs

data VariantRecord3
  = VariantRecord3A
      { variantRecord3A :: Int,
        variantRecord3B :: String,
        variantRecord3C :: Bool
      }
  | VariantRecord3B
      { variantRecord3C :: Bool,
        variantRecord3A :: Int,
        variantRecord3B :: String
      }
  | VariantRecord3C
      { variantRecord3B :: String,
        variantRecord3C :: Bool,
        variantRecord3A :: Int
      }
  deriving (Show)
```

```haskell
-- NextVersion.hs

data VariantRecord3
  = VariantRecord3C
      { variantRecord3A :: NewInt,
        variantRecord3C :: NewBool,
        variantRecord3B :: NewString
      }
  | VariantRecord3A
      { variantRecord3C :: NewBool,
        variantRecord3B :: NewString,
        variantRecord3A :: NewInt
      }
  | VariantRecord3B
      { variantRecord3B :: NewString,
        variantRecord3A :: NewInt,
        variantRecord3C :: NewBool
      }
  deriving (Show)
```

The latter type is the former type with the constructor and the field names reordered and with types replaced with new versions.

If we have the following instances:

```
instance Transformable Int NewInt
instance Transformable String NewString
instance Transformable Bool NewBool
```

Then, we can automatically generate a value of type `Transformer PreviousVersion.VariantRecord3 NextVersion.VariantRecord3` using a `genericReorderingTransformer` value provided by our state transformation library.

Using the `generics-sop` library, we can consider the codes, constructor names, and field names of both types:

```
Code PreviousVersion.VariantRecord3 =
  '['[Int, String, Bool], '[Bool, Int, String], '[String, Bool, Int]]
Code NextVersion.VariantRecord3 = '[
  '[NewInt, NewBool, NewString],
  '[NewBool, NewString, NewInt],
  '[NewString, NewInt, NewBool]]

ConstructorNamesOf PreviousVersion.VariantRecord3 =
  '["VariantRecord3A", "VariantRecord3B", "VariantRecord3C"]
ConstructorNamesOf NextVersion.VariantRecord3 =
  '["VariantRecord3C", "VariantRecord3A", "VariantRecord3B"]

FieldNamesOf PreviousVersion.VariantRecord3 = '[
  '["variantRecord3A", "variantRecord3B", "variantRecord3C"],
  '["variantRecord3C", "variantRecord3A", "variantRecord3B"],
  '["variantRecord3B", "variantRecord3C", "variantRecord3A"]]
FieldNamesOf NextVersion.VariantRecord3 = '[
  '["variantRecord3A", "variantRecord3C", "variantRecord3B"],
  '["variantRecord3C", "variantRecord3B", "variantRecord3A"],
  '["variantRecord3B", "variantRecord3A", "variantRecord3C"]]
```

Our transformer first converts a value between the following types:

```
type FirstType = PreviousVersion.VariantRecord3
type SecondType =
  SOP I '['[Int, String, Bool], '[Bool, Int, String], '[String, Bool, Int]]
type ThirdType =
  SOP I '['[Int, Bool, String], '[Bool, String, Int], '[String, Int, Bool]]
type FourthType =
  SOP I '[
    '[NewInt, NewBool, NewString],
    '[NewBool, NewString, NewInt],
    '[NewString, NewInt, NewBool]
    ]
type FifthType = NextVersion.VariantRecord3
```

The first and final steps are handled by the `generics-sop` library, and the penultimate step is done by our generic transformer. We need to implement the transformation between `SecondType` and `ThirdType`. The resulting type is completely new, so we must somehow infer it given the provided types `FirstType` and `FifthType`.

From `PreviousVersion.VariantRecord3` and `NextVersion.VariantRecord3`, we must infer the type-level list of type-level lists `'['[Int, Bool, String], '[Bool, String, Int], '[String, Int, Bool]]`. We use a type class to do this:

```
class
  DatatypesMatchReordering
    (a :: Type)
    (b :: Type)
    (cs :: [[Type]])
  where
  reorderConstructors :: SOP f (Code a) -> SOP f cs
```

By using associated type families, we can infer the type parameter `cs` from `a` and `b`. With the types we produce, we can manipulate the product and sum types to define corresponding term-level code that reorders SOPs.

The type-level logic can be represented with Prolog-like rules. The following facts define the properties of the types that we need:

```
%%% code(+T, -C) succeeds if C is the code of type T.
code(
  previous,
  [[int, string, bool], [bool, int, string], [string, bool, int]]
).
```

```
%%% constructor_names(+T, -C) succeeds if C is the list of constructor
%%% names of type T.
constructor_names(previous, [a, b, c]).
constructor_names(next,     [c, a, b]).

%%% field_names(+T, -C) succeeds if C is the list of field names of type
%%% T.
field_names(previous, [[a, b, c], [c, a, b], [b, c, a]]).
field_names(next,     [[a, c, b], [c, b, a], [b, a, c]]).
```

Our method involves first reordering the constructors to match and then reordering the fields. We must carry the field names along with the constructors. To do this, we zip symbols with lists of symbols with the `SymbolWithSymbols` type. Type families for type-level data of this kind are represented as follows:

```
%%% zip_symbols_with_symbols(+Symbols, +WithSymbolss, -SymbolWithSymbolss)
%%% succeeds if SymbolWithSymbolss is a list of
%%% symbol_with_symbols(Symbol, WithSymbols) zipping Symbols with
%%% WithSymbolss.
zip_symbols_with_symbols([], [], []).
zip_symbols_with_symbols(
  [Symbol|Symbols],
  [WithSymbols|WithSymbolss],
  [symbol_with_symbols(Symbol, WithSymbols)|SymbolWithSymbolss]
) :- zip_symbols_with_symbols(Symbols, WithSymbolss, SymbolWithSymbolss).

%%% zip_symbols_with_no_symbols(+Symbols, -SymbolWithSymbolss) succeeds
%%% if SymbolWithSymbolss is a list of symbol_with_symbols(Symbol, [])
%%% mapping Symbols.
zip_symbols_with_no_symbols([], []).
zip_symbols_with_no_symbols(
  [Symbol|Symbols],
  [symbol_with_symbols(Symbol, [])|SymbolWithSymbolss]
) :- zip_symbols_with_no_symbols(Symbols, SymbolWithSymbolss).

%%% symbols(+SymbolWithSymbolss, -Symbols) succeeds if Symbols is a list
%%% of symbols found in a list of symbol_with_symbols(Symbol, WithSymbols)
%%% SymbolWithSymbolss.
symbols([], []).
symbols(
  [symbol_with_symbols(Symbol, _)|SymbolWithSymbolss],
  [Symbol|Symbols]
) :- symbols(SymbolWithSymbolss, Symbols).

%%% with_symbols(+SymbolWithSymbolss, -WithSymbolss) succeeds if
%%% WithSymbolss is a list of lists of associated symbols found in a
%%% list of symbol_with_symbols(Symbol, WithSymbols) SymbolWithSymbolss.
```

```
with_symbols([], []).
with_symbols(
  [symbol_with_symbols(_, WithSymbols)|SymbolWithSymbolss],
  [WithSymbols|WithSymbolss]
) :- with_symbols(SymbolWithSymbolss, WithSymbolss).
```

We reorder a type-level list of types of kind `SymbolWithSymbols` according to a list of symbols, along with a type-level list of types found in a type's associated `Code`. We use the `OrderWithSymbols` and `TakeWithSymbols` type classes for this, represented as follows:

```
%%% take_with_symbols(+Types, +SymbolWithSymbolss, +SymbolToFind,
%%% ?FoundType, ?FoundSymbolWithSymbols, ?RemainingTypes,
%%% ?RemainingSymbolWithSymbolss) succeeds if we find type FoundType and
%%% symbol with symbols FoundSymbolWithSymbols when searching
%%% SymbolWithSymbolss zipped with Types for SymbolToFind. Also,
%%% RemainingTypes and RemainingSymbolWithSymbolss must be the remaining
%%% types and symbols with symbols.
take_with_symbols(
  [Type|Types],
  [symbol_with_symbols(Symbol, WithSymbols)|SymbolWithSymbolss],
  SymbolToFind,
  FoundType,
  FoundSymbolWithSymbols,
  RemainingTypes,
  RemainingSymbolWithSymbolss
) :-
  (Symbol = SymbolToFind -> SymbolsAreEqual = true; SymbolsAreEqual = false),
  take_with_symbols_with_predicate(
    SymbolsAreEqual,
    [Type|Types],
    [symbol_with_symbols(Symbol, WithSymbols)|SymbolWithSymbolss],
    SymbolToFind,
    FoundType,
    FoundSymbolWithSymbols,
    RemainingTypes,
    RemainingSymbolWithSymbolss
  ).

take_with_symbols_with_predicate(
  true,
  [Type|Types],
  [symbol_with_symbols(Symbol, WithSymbols)|SymbolWithSymbolss],
  SymbolToFind,
  Type,
  symbol_with_symbols(Symbol, WithSymbols),
  Types,
  SymbolWithSymbolss
```

```
).
take_with_symbols_with_predicate(
  false,
  [Type1,Type2|Types],
  [SymbolWithSymbols1, SymbolWithSymbols2|SymbolWithSymbolss],
  SymbolToFind,
  FoundType,
  FoundSymbolWithSymbols,
  [Type1|RemainingTypes],
  [SymbolWithSymbols1|RemainingSymbolWithSymbolss]
) :-
  take_with_symbols(
    [Type2|Types],
    [SymbolWithSymbols2|SymbolWithSymbolss],
    SymbolToFind,
    FoundType,
    FoundSymbolWithSymbols,
    RemainingTypes,
    RemainingSymbolWithSymbolss
  ).

%%% order_with_symbols(+Types, +SymbolWithSymbolss, ?Symbols, ?NewTypes,
%%% ?NewSymbolWithSymbolss) succeeds if NewTypes and
%%% NewSymbolWithSymbolss are produced after reordering Types and
%%% SymbolWithSymbolss zipped together according to Symbols.
order_with_symbols(Types, [], [], Types, []).
order_with_symbols(
  [Type|Types],
  [SymbolWithSymbols|SymbolWithSymbolss],
  [Symbol|Symbols],
  [NewType|NewTypes],
  [NewSymbolWithSymbols|NewSymbolWithSymbolss]
) :-
  take_with_symbols(
    [Type|Types],
    [SymbolWithSymbols|SymbolWithSymbolss],
    Symbol,
    NewType,
    NewSymbolWithSymbols,
    RemainingTypes,
    RemainingSymbolWithSymbolss
  ),
  order_with_symbols(
    RemainingTypes,
    RemainingSymbolWithSymbolss,
    Symbols,
    NewTypes,
    NewSymbolWithSymbolss
  ).
```

The `OrderWithSymbols` type class can be used to provide a method that reorders a product of types. This is possible as a product is represented as a list. Sums of types are represented as Peano numbers and are therefore much harder to reorder. To reorder sums, we use *injections*, functions that produce sums. We first use `OrderWithSymbols` to find the desired type-level lists for our output. This allows us to create a product of injections that produce the values of the new sum's type. Then, we use `OrderWithSymbols` again to reorder this product of injections such that they can be applied with the input sum. The constraints for this are represented as follows:

```
order_with_symbols_ns(
  [Type|Types],
  [SymbolWithSymbols|SymbolWithSymbolss],
  [Symbol|Symbols],
  [NewType|NewTypes],
  [NewSymbolWithSymbols|NewSymbolWithSymbolss]
) :-
  order_with_symbols(
    [Type|Types],
    [SymbolWithSymbols|SymbolWithSymbolss],
    [Symbol|Symbols],
    [NewType|NewTypes],
    [NewSymbolWithSymbols|NewSymbolWithSymbolss]
  ),
  symbols([SymbolWithSymbols|SymbolWithSymbolss], OriginalSymbols),
  order_with_symbols(
    [NewType|NewTypes],
    [NewSymbolWithSymbols|NewSymbolWithSymbolss],
    OriginalSymbols,
    [Type|Types],
    [SymbolWithSymbols|SymbolWithSymbolss]
  ).
```

We also need a type class that reorders products found inside sums, once the sums have been reordered according to constructor names. This is represented as follows:

```
order_with_symbols_nps([], [], [], []).
order_with_symbols_nps(
  [Types|Typess],
  [WithSymbols|WithSymbolss],
  [Symbols|Symbolss],
  [NewTypes|NewTypess]
) :-
  zip_symbols_with_no_symbols(WithSymbols, WithSymbolsWithNoSymbols),
  order_with_symbols(
    Types,
    WithSymbolsWithNoSymbols,
    Symbols,
```

```
    NewTypes,
    _
  ),
  order_with_symbols_nps(Typess, WithSymbolss, Symbolss, NewTypess).
```

We can now define an instance of `DatatypesMatchReordering` that infers type variables correctly:

```
%%% datatypes_match_reordering(+T1, +T2, ?NewCode) succeeds if NewCode is
%%% the code of T1 reordered by matching the constructor and field names
%%% of T1 to those of T2.
datatypes_match_reordering(T1, T2, NewCode) :-
  code(T1, Code1),
  constructor_names(T1, ConstructorNames1),
  constructor_names(T2, ConstructorNames2),
  field_names(T1, FieldNames1),
  field_names(T2, FieldNames2),
  zip_symbols_with_symbols(
    ConstructorNames1,
    FieldNames1,
    ConstructorNamesWithRecordNames1
  ),
  order_with_symbols_ns(
    Code1,
    ConstructorNamesWithRecordNames1,
    ConstructorNames1,
    ConstructorReorderedCode,
    ConstructorReorderedConstructorNamesWithRecordNames
  ),
  with_symbols(
    ConstructorReorderedConstructorNamesWithRecordNames,
    ConstructorReorderedRecordNames
  ),
  order_with_symbols_nps(
    ConstructorReorderedCode,
    ConstructorReorderedRecordNames,
    FieldNames2,
    NewCode
  ).
```

The following query uses this relation to infer the correct reordering for the original types.

```
?- datatypes_match_reordering(previous, next, Code).
Code = [[int, bool, string], [bool, string, int], [string, int, bool]].
```

C Proposal

Practical Dynamic Software Updating for Haskell

Part II Computer Science Project Proposal

October 13, 2022

Project Originator:	**The Dissertation Author**
Project Supervisor:	**Mistral Contrastin**
Director of Studies:	**(name removed)**
Overseers:	**Alan Blackwell, Srinivasan Keshav**

Introduction and Description

Dynamic Software Updating (DSU) is a technique that involves patching programs while they are running, without changing their state. DSU is typically either used in production, for long-running applications, or in development, for live programming systems [1]. We will focus on long-running programs, including web servers, database management systems, and operating systems.

For web servers, DSU is useful for fixing bugs and adding features without interrupting service. This provides an alternate approach to zero downtime deployment, which is typically handled with expensive and complex redundancy. Non-redundant systems that do not require zero downtime deployment may also benefit from DSU when restarting them is disruptive, such as with desktop operating systems [2].

There are several approaches to DSU that ensure that updates to programs are safe and correct. These approaches typically employ timing restrictions that determine when updates can take place. Timing restrictions can either be implemented automatically or manually, where the programmer performs manual identification of legal update points. Systems like OPUS [3] automatically enforce activeness safety, where updated functions must not exist on the call stack at update time. Systems like Ginseng [4] automatically enforce cons-freeness safety, where updates can only take place where updated types are not accessed concretely. In an empirical study, Hayden et al. [5] found that, in terms of number of permitted failures and programmer effort, the use of manual identification, following simple design patterns, was the most effective timing restriction.

The manual update point identification approach for DSU has been implemented in the Kitsune DSU system for C [6] and the Rubah DSU system for Java [7]. These systems have successfully updated a number of long-running, open source programs, including Redis (a key-value database written in C) and JavaEmailServer (an POP3/SMTP server written in Java). Both systems impose no noticeable performed overhead on normal execution, require little programmer effort, and are flexible enough to allow almost any update.

Kitsune and Rubah take advantage of the structure of long-running programs, where an infinite loop handles global state. Function calls are added to these loops where it is safe to perform an update. Typically, it is safe to update at the start or end of one of these loops. When one of these points are reached, the new version of the program is loaded into memory and reaches the same point, where execution is continued. The global state (and a set of certain local variables chosen by the programmer) is transformed and transferred into the new program.

We intend to apply Kitsune and Rubah's approach to a functional language, Haskell. Functional programs typically appear different to C and Java programs, especially in terms of state. Data is preferred to be immutable, which reduces or completely removes the need for global variables. This is especially true in Haskell, which is purely functional. The core approach of Kitsune and Rubah's implementations should still be applicable, as the concepts of an infinite "loop" with a global state, required by all long-running programs, are still used in a functional style. However, the ways in which the DSU system integrates with the control flow and state management will need to be different.

In C, global variables are commonly used for state. In contrast, Haskell programs often use monads in the bodies of short tail-recursive functions for state. Our approach involves this state being specified in the function indicating the update point. Using a GHC plugin, the transformed state is made available, along with other data about the update, across new program versions. Program execution can skip to the update point and load the updated state by accessing this data.

One problem we expect to encounter is the type of state being polymorphic. In an update in this case, it may not be clear which type transformer would be applicable. We will investigate methods to solve this, including the manual removal of polymorphism, using an existing GHC plugin that performs monomorphisation [8], or using Haskell's reflection features.

To deal with the issue of Haskell's laziness, we are initially only allowing non-lazy updates to occur. This will mitigate issues with data representation, as serialisation can be used as a backup, but will limit the number of updates that can be made, as it would be difficult to immediately move certain infinite data structures from one representation to another. An extension would add support for lazy updates, which can be evaluated against the non-lazy approach.

To inject data everywhere across Haskell code, which has many forms of expression, we will make use of the Glasgow Haskell Compiler (GHC) Core language, also known as System FC. This is a lower level representation of Haskell code that is used by the GHC during compilation. This has far fewer constructs than Haskell, but still preserves enough meaning for it to be useful for DSU [9]. Using Core should reduce the number of edge cases that would be needed for data injection.

In Kitsune, new code is loaded at runtime using dynamic linking. In Rubah, an approach that rewrites bytecode is used instead. As Haskell is typically compiled to native machine code, we will use a dynamic linking approach. Data should be able to be transferred between different versions of a program without issues arising from different representations, but serialisation should be usable as a backup when this is not the case. In addition, GHCi, as used in Haskell DSU systems for live coding such as Tidal [10], can be used as a backup if dynamic linking fails.

We will evaluate this DSU using open source long-running programs with long commit histories, selected from GitHub. By ensuring that these programs all sufficiently varied, the behaviour of the DSU system on these programs should be representative of most

long-running Haskell programs. For this project to be successful, the DSU system should be able to perform many updates, represented by commits, from samples of the commit histories of the programs being updated. In addition, the steady state performance of programs, in terms of memory and speed, using DSU system will be characterised, as well as update time.

A more detailed description of the work to be carried out is given in the Structure of the Project section.

Starting Point

I have no experience implementing DSU systems. I know the basics of Haskell, having completed a Functional Programming in Haskell course in 2020, but I have no experience with real-world Haskell programs. I also have no experience with the GHC Core language or writing GHC plugins. I have functional programming knowledge from OCaml programming at Jane Street.

Prior to starting, I have read literature on Ginseng [4], Kitsune [6], and Rubah [7] DSU systems. I have some general knowledge of the low level concepts of intermediate representations and data representation from Compiler Construction.

I have also already identified some candidate Haskell programs that I could implement DSU for, including xmonad (a tiling window manager), hnes (an NES emulator), and tart (a terminal art program).

Structure of the Project

Core

The core of this project is composed of the following parts:

Simple DSU runtime. Implement a Haskell runtime system that can load new versions of a program into memory and unload old versions. Implement a system where this occurs after a function the API has been called and a signal such as SIGUSR2 has been received.

Data injection. Create a GHC compiler plugin that injects a parameter into all functions (in GHC Core form) that can be used to access information about previous updates. This part will use techniques from plugins like Eta [11].

Code injection. Extend the compiler plugin to inject the runtime. Implement parts of the update function that extract and import state, including in cases where the type of state is polymorphic. This part will use techniques from plugins like Evoke [12].

State transformation system. Implement a system that extracts state transformations from Haskell code. These transformations convert state from one type to another. Extend the compiler plugin to inject these transformations, using techniques from plugins like InterpolatePlugin [13].

Further API. Add functions to the API that give information about the last update, similar to those found in Kitsune.

Build system. Implement a build system that takes a directory structure with different versions of a program and Haskell state transformations. This build system will give an object file for the original version of the program and object files that can be dynamically linked in by the original version's DSU runtime to successfully update the program.

Evaluation

In order to evaluate the above, we will:

- Find a number of long-running Haskell programs that are used for a range of purposes.
- Take a sample of versions for each program and set them up to be updated by the DSU system.
- Determine the number of updates that are possible, and assess the correctness of the updates.
- For a sample of program versions, measure the effect on performance, in terms of speed and memory, of the program after making it compatible with DSU and after performing updates. Also, measure the update time.

Extensions

The following extension is possible:

Lazy state transformations. Add the option to perform state transformations lazily rather than only when updates occur. This would increase the number of updates that can be performed, but could negatively affect the performance of programs after chains of updates. This approach would be evaluated against the non-lazy approach.

Success Criteria

Our success criterion is as follows:

A successful DSU system for long-running programs should be able to perform many updates on real-world programs.

Timetable and Milestones

Weeks 1 to 2 (6th October – 19th October)

- *Submit project proposal.*
- Write the introduction chapter.
- Begin working on the simple DSU runtime part.
- Begin writing the implementation chapter while programming.

Weeks 3 to 4 (20th October – 2nd November)

- Finish implementing the simple DSU runtime part.
- Begin writing the preparation chapter, including the requirements analysis.
- Begin writing the data injection part.

Weeks 5 to 6 (3rd November – 16th November)

- Finish writing the data injection part.
- Begin writing the code injection part.

Weeks 7 to 8 (17th November – 30th November)

- Finish writing the preparation chapter.
- Finish writing the code injection part.
- Begin writing the state transformation system part.

Weeks 9 to 10 (1st December – 14th December)

- *End of Michaelmas term.*
- Take time off for Christmas.
- Finish writing the state transformation system part.

Weeks 11 to 12 (15th December – 28th December)

- Take time off for Christmas.

Weeks 13 to 14 (29th December – 11th January)

- Write the further API.
- Begin writing the build system.

Weeks 15 to 16 (12th January – 25th January)

- *Start of Lent term.*
- Finish writing the build system.
- Apply DSU to a real-world program.

Weeks 17 to 18 (26th January – 8th February)

- *Create and submit progress report.*
- Fix bugs relating to the DSU being applied to the real-world program and improve the API depending on needs.

Weeks 19 to 20 (9th February – 22nd February)

- *Give progress report presentation.*
- Apply DSU to another real-world program.
- Start improving the implementation chapter written so far.
- Start writing the evaluation chapter.

Weeks 21 to 22 (23rd February – 8th March)

- Evaluate the DSU system on the first real-world program.
- Apply DSU to two real-world programs.

Weeks 23 to 24 (9th March – 22nd March)

- *End of Lent term.*
- Evaluate all the real-world programs.
- Finish the evaluation chapter.

Weeks 25 to 26 (23rd March – 5th April)

- Write conclusions chapter.
- Implement the lazy state transformations extension if time permits.

Weeks 27 to 28 (6th April – 19th April)

- Implement the lazy state transformations extension if time permits.

Weeks 29 to 30 (20th April – 3rd May)

- *Start of Easter term.*
- Work on improving the dissertation, with supervisor feedback.

Weeks 31 to 32 (4th May – 12th May)

- *Submit dissertation.*

Resource Declaration

I will write my dissertation and code on my personal laptop (1.80 GHz i7-10510U, 8 GB RAM, 500 GB SSD). *I accept full responsibility for this machine and I have made contingency plans to protect myself against hardware and/or software failure.*

As a backup, I will use College and Department computers. I will frequently back up my dissertation and code with Git-based version control on GitHub.

References

[1] Pablo Tesone et al. "Dynamic software update from development to production". In: *The Journal of Object Technology* 17.1 (2018), pp. 1–36.

[2] Michael Hicks and Scott Nettles. "Dynamic software updating". In: *ACM Transactions on Programming Languages and Systems* 27.6 (Nov. 2005), pp. 1049–1096. ISSN: 0164-0925. DOI: 10.1145/1108970.1108971.

[3] Gautam Altekar et al. "OPUS: online patches and updates for security". In: *14th USENIX Security Symposium (USENIX Security 05)*. Baltimore, MD: USENIX Association, July 2005.

[4] Iulian Neamtiu et al. "Practical dynamic software updating for C". In: *Proceedings of the ACM Conference on Programming Language Design and Implementation (PLDI)*. Ottawa, Canada, June 2006, pp. 72–83.

[5] Christopher M. Hayden et al. "Evaluating dynamic software update safety using efficient systematic testing". In: *IEEE Transactions on Software Engineering* 38.6 (Dec. 2012). Accepted September 2011, pp. 1340–1354. DOI: 10.1109/TSE.2011.101.

[6] Christopher M. Hayden et al. "Kitsune: efficient, general-purpose dynamic software updating for C". In: *Proceedings of the ACM Conference on Object-Oriented Programming Languages, Systems, and Applications (OOPSLA)*. Oct. 2012.

[7] Luís Pina, Luís Veiga, and Michael Hicks. "Rubah: DSU for java on a stock JVM". In: *Proceedings of the ACM Conference on Object-Oriented Programming Languages, Systems, and Applications (OOPSLA)*. Oct. 2014.

[8] Conal Eliott. *Monomorph*. https://github.com/conal/monomorph.

[9] Richard A. Eisenberg et al. *System FC, as implemented in GHC*. URL: https://github.com/ghc/ghc/raw/463ffe0287eab354a438304111041ef82d2ed016/docs/core-spec/core-spec.pdf (accessed on 5 Oct. 2022).

[10] Alex McLean et al. *Tidal*. https://github.com/tidalcycles/Tidal.

[11] Takano Akio and Mitsutoshi Aoe. *Eta*. https://github.com/takano-akio/eta-plugin.

[12] Taylor Fausak. *Evoke*. https://github.com/tfausak/evoke.

[13] Danil Berestov. *InterpolatePlugin*. https://github.com/goosedb/InterpolatePlugin.