Type-Correct Changes — A Safe Approach to Version Control Implementation

Jason Dagit

November 13, 2010

Contents

Chapter 1

Introduction

Version control systems require a high degree of robustness as users trust them to safeguard their data over the life cycle of software projects. Corruption in repository data, such as the history of changes, can lead to wasted time and user frustration. Worse yet is the possibility of a bug which constructs invalid versions of the user's data. When the data under version control is source code this can lead to build failures or forms of corruption that go unnoticed until it poses a problem.

Software engineers have many tools to help write software applications. Many of these tools exist to tackle the challenge of writing correct software. Testing continues to be a popular approach to this challenge but testing is not usually enough to prove correctness. Instead, software engineers use testing to gain confidence that the tested program will behave as intended in most uses.

When testing is not sufficient, formal methods may be used to prove parts of the program correct. Often formal methods are applied in only the core of applications due to the high labor costs needed to use them effectively. Another approach to reducing the high cost of formal methods is to use an automated method such as a proof assistant. Proof assistants are only available in specialized domains, such as research programming languages. This means that many mainstream programming environments lack automated proof tools.

Given the importance of correctness for version control systems, we would like to eliminate as many bugs as possible from the Open Source version control system Darcs [Rou09a]. We examine the data model used by Darcs and discuss a number of invariants that must be maintained to avoid data corruption.

Darcs is implemented in the programming language Haskell, which gives us an opportunity to apply modern innovations in Programming Language research to a real-world software application. Our approach shows that the type system of the Haskell programming language together with a novel combination of language extensions, implemented by the Glasgow Haskell Compiler (GHC), can be used as a light-weight alternative to a proof assistant. We show that entire classes of bugs can be eliminated at compile time using our approach.

Finally we examine the impact of our techniques on the Darcs codebase and the challenges that arise when applying these techniques to an existing real-world application.

1.1 Background

This document shows how to encode specific invariants into types in the Haskell programming language. Although program invariants may seem unrelated to types, we use properties of Haskell's strong static type checking to gain static guarantees about these invariants.

1.1.1 Patch Theory

The data model used and pioneered by Darcs is known as *Patch Theory* [Rou09b], discovered by David Roundy to solve the problem of communicating changes in a distributed fashion between contributors. Patch Theory remains distinct in that it allows users to think of their repositories as unordered collections of changes. More details can be found in Chapter 3.

1.1.2 Haskell's Type System

We assume the reader has a basic familiarity with functional programming and the Haskell language specifically. For more details about the language features that will be discussed in this document please see Section 2.2 and Appendices A and B.

Haskell's type system is based on Hindly-Milner type checking [Pey03]. The GHC implementation of Haskell uses a modified version of the Damas-Milner type checking algorithm [VWP06]. In fact, GHC contains many extensions compared to the language specification for Haskell [Pey03] and it is no coincidence that GHC is used as the compiler for Darcs. Many of the extensions implemented by GHC are of great value for real-world Haskell programming.

1.2 Motivation

The use of version control systems (VCS) seem to be a common practice for software projects these days as most projects use some form of version control. Example version control systems include Subversion (SVN) [Tig09], Concurrent Versions System (CVS) [Fre09], Git [Tor09], Darcs [Rou09a], BitKeeper [Bit09], Monotone [Mon09], Visual SourceSafe [Mic09], and many others.

A VCS plays a support role in a project. That is, the VCS used by a team of software developers supports the primary task of software development. For this reason it is important that the VCS be reliable and robust, otherwise the software developers could lose time dealing with their tools instead of working on their primary task of software development.

Robustness has always been important for Darcs and it has motivated the Darcs project to try new things, such as moving the implementation language from C++ to Haskell as explained by Roundy [Sto05]:

> It is a little-known fact that the first implementation of Darcs was actually in C++. However, after working on it for a while, I had an essentially solid mass of bugs, which was very hard to track down.

While Darcs has a test suite that continues to grow in size and comprehensiveness, it does not provide a total solution for ensuring the level of quality assurance that users demand. As an illustration of this point, we include code coverage statistics generated by the Haskell Program Coverage (HPC) toolkit [GR07] in Appendix D. The number of bugs in Darcs, despite testing, is a strong motivating factor in our decision to incorporate more proof techniques in our quality assurance process.

1.3 Structure of this document

This document introduces the related work in Chapter 2 in the areas of automated invariant checking and encoding invariants in Haskell programs. In Chapter 3 we will give the necessary background for understanding several key invariants of Darcs. The tools and abstractions we use to represent these invariants, along with real examples are given in Chapter 4. Our analysis of the work, with discussion is found in Chapter 5. Finally we give a closing statement in Chapter 6.

Chapter 2

Related Work

In this chapter, we present work related to this thesis. A brief survey of the version control landscape is given in Section 2.1. An overview of type-based proofs as well as proof-carrying types can be found in Section 2.2.

2.1 Version Control Systems

Version control systems (VCS) are used by many software developers, projects and organizations. The primary feature offered by VCS software is the ability to track modifications to a collection of documents, usually program source code. Typically users are allowed to make their modifications independently and then share the modifications. Common VCS operations are covered in Section 2.1.1. A common classification among VCS is whether the modifications are shared in a distributed or centralized fashion. This distinction and where Darcs fits is covered in Section 2.1.2.

2.1.1 Commonly Supported Features

Every VCS has a notion of modification although different terminology is often used such as change, patch, or revision. The VCS stores a collection of documents along with the history of modifications in what is known as a repository.

The first step for using a VCS is usually to get a copy of a repository where the user can make modifications. Once modifications have been made, the VCS requires the user to record, or commit, the modifications. Doing so creates an entry in the history that contains the changes and usually a description entered by the user. After making and recording modifications users will often need to share their work with others who have a copy of the repository. The different ways of sharing are covered in the next section.

An important feature of most VCS is that of branching and merging. Creating a branch means making a copy of the repository which diverges from the original repository. In a software development project this might be done to facilitate the design and development of an experimental new feature while applying bug fixes to a stable version. Following this example, once the new feature is complete we would like to merge the two repositories so that we are left with one repository with both the completed new feature and the bug fixes to existing functionality. How branching and merging function also depends on the distinction between centralized and decentralized version control.

2.1.2 Centralized and Decentralized Version Control

How to share the changes, the ways in which they can be shared, and the order that they can be shared varies between VCS. Many VCS require that there is a central repository which collects all the changes and users connect to it to share and receive changes. Some VCS allow changes to be shared directly between copies of the repository in a decentralized fashion.

Well known examples of centralized VCS include, Subversion (SVN) [Tig09], Concurrent Versions System (CVS) [Fre09], Perforce [Per09], and Visual SourceSafe [Mic09]. Each of these VCS operate in client-server manner. The central repository is the server and each user has a client repository which communicates only with the central repository.

Decentralized, also known as distributed, VCS allow repositories to communicate directly removing the client and server distinction found in centralized VCS. Well known examples of decentralized VCS include, Darcs [Rou09a], Mercurial [Sel09], Git [Tor09], and Bazaar [Can09].

Modifications made with a centralized VCS may be stored in the order that they are committed to the central repository. This provides a natural linear progression of modifications and typically forces an implicit dependency between the modifications. Generally, new modifications must be made on top of all previous modifications. For example with SVN, users typically must update their local repository with modifications from the central repository before committing new changes.

With decentralized VCS the task of sharing changes becomes more complex as it is often equivalent to merging two repositories. For example, in the Darcs data model each copy of a repository is considered a branch and every time patches are shared it is equivalent to a merge in the SVN data model. In fact, these *spontaneous branches* set Darcs apart even within the category of decentralized VCS. Other decentralized VCS, such as Git, store modifications in a specific order whereas Darcs allows the order of modifications to be reordered according to the rules of Patch Theory discussed in Chapter 3.

2.2 Type Level Proofs

Although our implementation work is done inside of Darcs, our focus is not on the VCS aspects. Instead we are focused on using the type system as theorem prover and proof assistant. We discuss Haskell based type level proofs in Section 2.2.1. Briefly we discuss type system based proofs in mainstream languages and dependently typed languages in Section 2.2.2.

2.2.1 Haskell

Here we focus on proofs and proof techniques based in Haskell's type system. Much of the research in Haskell that uses the type system for proofs centers around the use of type classes. This may be due in part to how long type classes have been available in Haskell and their standardization. More recent work in this area has demonstrated the power of Generalized Algebraic Data Types (GADTs). Appendix B contains a brief overview of GADTs and examples involving GADTs can be found in Chapter 4.

Language Features

The Haskell programming language [Pey03] specifies Hindley-Milner type inference and checking. Hindley-Milner type inference combined with type classes, nested types and recursive types gives Haskell programmers a plethora of interesting and useful idioms and techniques. Some of the techniques and idioms discussed in the research allow the Haskell type checker to serve as a proof assistant at compile time.

In addition to the features above, we focus on several other features supported by the Haskell compiler GHC [GHC09c]:

- Generalized Algebraic Data Types (GADTs), developed by Xi et al, Jones et al, and Cheney and Hinze [XCC03, PVWW06, CH03];
- Existentially quantified types [LO94], commonly referred to as *existential types*, explained in Appendix A, and;
- Phantom Types [LM99].

We have two main uses for existential types. First, we borrow the idea of branding [KS07] when we need a type that is distinct and; second, to express certain type relations in our data types without exposing the exact types in the type of the data structure.

Both existentially quantified types and phantom types are implied by using GADTs, but our usage of them is important enough to warrant introducing them separately.

Some authors, such as Baars and Swierstra, have used the term *witness type* to refer to a type that serves as a witness of a proof. For example the type could represent a proof that two types are equal [BS02]. We adopt this terminology in our work.

Witness types are chiefly useful to us as a means of ensuring certain invariants are preserved. In the case of Darcs we would like to be able to change the semantics, fix bugs or refactor the code and always know that the properties of Patch Theory, such as those discussed in Chapter 3, have been respected.

Peyton-Jones et al [PVWS07] extended the type system used by GHC to handle arbitrary rank, which leads to so called "sexy types." Sexy types include higher rank polymorphism and existential types. Additionally, sexy types give us precisely the power we need to express run-time invariants through the type system as demonstrated by Shan [Sha04]:

> ... skillful use of sexy types can often turn what is usually regarded as a run-time invariant into a compile-time check. To implement such checks is to reify dynamic properties of values as refined distinctions between types. These distinctions in turn increase the degree of heterogeneity among types in the program.

Hinze shows that higher rank types can also be used to enforce a wide variety of invariants in data types [Hin01].

Using existential types Kahrs shows us how to encode the invariants of red-black trees [Kah01]. The existential types are used in the data type declaration to control unification of phantom types. We use a similar means to control unification of phantom types in our implementation. As Kahrs mentions, using phantom types in this way has the advantage that it can be removed later, once the code is known to preserve invariants and the phantom types add no run-time cost.

Type Class Based

Using type classes it is possible to implement a statically checked run-time test for type equality using witness types as detailed by Baars and Swierstra [BS02]. Implementing type equality this way does have one draw back as demonstrated by Kiselyov [Kis09], namely it is possible to weaken the type system through malicious type class instances. In Section 5.2.3, we discuss a similar problem that threatens the context equality that we use in the Darcs implementation.

Type classes, especially when combined with functional dependencies, allow for computations in the type system as explained by Hallgren [Hal01]. Any purely functional computation that terminates appears to be possible at the type level. For example, basic arithmetic on type level natural numbers is relatively easy to express. One drawback to this variety of type level proof is that by enabling this level of computation in the type system we lose the property that type checking will always terminate.

Kiselyov and Shan [KS04] provide a powerful example of how Haskell's type classes can be used to turn values into types and back again. These authors give a way to reify any value that can be serialized into the type system. A major drawback of using this approach is that it adds run-time overhead. Converting back and forth between types and values requires processing overhead and there is also the overhead of passing run-time data for each type. The run-time overhead can be proportional to the "size" of the type [McB02]. Our implementation is already burdened by performance issues and so we seek to avoid adding any additional run-time overhead.

Silva and Visser [SV06] give another great example of Haskell programmers reaching for more static safety by exploiting the types system and HList [KLS04]. As Silva and Visser describe their work:

> We explain how type-level programming can be exploited to define a strongly-typed model of relational databases and operations on them. In particular, we present a strongly typed embedding of a significant subset of SQL in Haskell. In this model, meta-data is represented by type-level entities that guard the semantic correctness of database operations at compile time.

By using HList, values with heterogeneous types may be stored together in a record, or list, of arbitrary size. While this is similar to our Directed Lists, see Section 4.6, we would like to place more constraints on our data types such as Hinze [Hin01] does and also not exposing the intermediate types of the elements in our directed types.

The libraries Dimensionalized Numbers [Den09] and Dimensional [Buc09] both take the approach of exposing extra information to the type system to achieve correct unit manipulations. In both of these cases the correctness the authors want to model is that arithmetic operations should respect the physical units involved.

GADT Based

Eaton [Eat06] gives a clever way to expose matrix dimensionality to the type system so that only operations which respect the dimensions of arrays and matrices statically are allowed. This approach is interesting because it is not unlike our own and yet only uses GADTs incidentally. Meaning, it is not a core requirement for their approach. As the author says the technique is to "expose certain properties of operands to a type system, so that their consistency could be statically verified by a type checker, then we would be able to catch many common errors at compile time."

The presented approach uses type classes and the type reflection technique presented by Kiselyov and Shan [KS04]. Similar to our experiences, this author also points out that doing so increases the type signatures in an unpleasant way. A major difference between our implementation and that of Eaton is the use of functional dependencies [Jon00]. Functional dependencies allow the programmer to place constraints on the types used in a type class. If our approach relied on type classes we would probably use functional dependencies as well. A minor difference between our approaches is that while we use data types with existentially quantified types as wrappers so that we may have existential types result from functions Eaton prefers to use CPS transformation. This transformation leads to equivalent types [Sha04, Eat06]. Eaton also notices how type checking is now so difficult as to be a burden to the programmer and comments that data flow analysis may be able to improve type check error messages. Such an improvement by any means would be very welcome.

Greif [Gre08] applies the same data declarations that we use for directed lists, Section 4.6, to implement *Thrists*, or type threaded lists. Although this work is unpublished, according to the author it is inspired by the brainstorming session at *Haskell'05* workshop in Tallinn. This session is where our directed lists were born. Greif provides a library for Thrists in both Ωmega and Haskell, with several example applications including parsers and interpreters.

Faking Dependent Types

Although we do not use a dependently typed language for our implementation, we do approximate, or simulate, dependent typing within Haskell to achieve some of our goals. McKinna [McK06] explains the benefits of dependently typed programming:

> Type systems without dependency on dynamic data tend to satisfy the replacement property—any subexpression of a well typed expression can be replaced by an alternative subexpression of the same type in the same scope, and the whole will remain well typed. For example, in Java or Haskell, you can always swap the then and else branches of conditionals and nothing will go wrong—nothing of any static significance, anyway. The simplifying assumption is that within any given type, one value is as good as another. These type systems have no means to express the way that different data mean different things, and should be treated accordingly in different ways. That is why dependent types matter.

This observation exactly characterizes why Darcs became fragile and why we seek to simulate dependent typing. Replacing patches in, concatenating and rearranging patch sequences was always statically valid even when it would result in corrupt repositories. For this reason we sought out techniques that would give us the benefits of dependent typing in Haskell.

Using type level numerals, Fridlender and Indrika [FI00] show a simple way to work around the lack of dependent types in Haskell. The main example given allows us to make a version of the standard Haskell function `zipWith`, which is referred to as `zipWithN`, that is type indexed by a type level numeral. The numeral represents the number of parameter lists passed to `zipWithN`. This approach is representative of simulating dependent typing with Haskell. One type is created for each value, in this case type level numerals. To simulate the values inhabiting a type we can make each type an instance of the same type class. Thus the values correspond to Haskell types and the types correspond to Haskell type classes.

McBride [McB02] explores the simulation of dependent typing in Haskell. This paper explains various tricks to simulate dependent typing and how they are related. It also clearly explains how type classes allow the programmer to simulate some type families. He also comments on the limitations of type inference and what can be accurately encoded when using nested types such as those used by Okasaki [Oka99]. McBride warns us that run-time overhead of type class heavy techniques may be proportional to the size of the type signatures. In the GHC implementation this results from implicit passing of type dictionaries for functions that rely on type classes.

Guillemette and Monnier [GM08] found that it was possible to represent subset and superset relationships in the type system using GADTs. This also required a way to implement type equality as a run-time test. Their techniques are very similar to ours even though the domain is very different, a type-preserving compiler. They use type level Peano numbers to represent de Bruijn indices.

Kiselyov and Shan [KS07] tag, or brand, values with types that represent certain capabilities. For example, by creating a new list datatype where the type of the list is parametrized by a brand we can statically enforce non-emptiness of lists. The brand is part of the type of the list and acts as a proof of a capability such as whether the list is empty or non-empty. The work done here is in OCaml but applies equally well to Haskell and can be used even without dependent typing, although this requires that we use a trusted kernel of code which may do run-time checks to generate the correct branding. Once we have the branding in place the type system can do the verification, thus we can restrict our intensive verification to just the trusted kernel. This is essentially the approach we have taken for directed lists. This work is also similar to the examples of dependent typing given by Xi [XP98]. Xi uses restricted dependent types to remove array bounds checking.

By using "nested types, polymorphic recursion, higher-order kinds, and rank-2 polymorphism," Okasaki [Oka99] is able to encode vector and matrix dimensions into types. This encoding ensures that matrix and vector operations can be statically checked for correctness.

2.2.2 Non-Haskell

Proving properties and carrying the proofs with types is not limited to Haskell. Skalka and Smith [SS00] propose a type system for statically enforcing security using the JVM security model. The type system carries proofs about the code as it is compiled. For

this to work static type inference is required, this means that their static security does not work without a modified Java compiler.

Java is not the only mainstream programming language that is receiving attention from type based proofs. Kennedy and Russo [KR05] have found a way to bring the power of GADTs to C# and Java. Hopefully in the future many of the approaches discussed here will apply in mainstream languages. Before we can freely use GADTs in C# the compiler would need to be augmented with the special type checking rules described.

Xi and Scott [XS99] make a very good survey of work done in dependent typing, give examples where it helps and explain why it is an important subject.

One example of using language features similar to GADTs arises in a dependently typed variant of ML known as Dependent ML. Chen and Xi [CX03] use Dependent ML to implement type correct program transformations.

Sheard [She05] explains, with examples, a Haskell-like language known as Ωmega. Ωmega has GADTs but unlike Haskell it offers strict evaluation and features designed to ease using the type system for proofs. Unfortunately we could not use Ωmega without a substantial rewrite of Darcs. It also not clear that Ωmega is ready for real-world use. We hope that the techniques we demonstrate help answer a question posed by the author about the way in which other features such as rank-n polymorphism magnify the benefit of GADTs.

Chapter 3

Data Model and Invariants

Now we turn to establishing the theory underlying Darcs. We assume the reader has basic familiarity with the use of version control systems. Here we describe the fundamentals of Patch Theory [Rou09c] as it relates to version control. Not all of Patch Theory has been made rigorous and precise at this time although Roundy has made several presentations on Darcs that include discussions of Patch Theory [Rou06a, Rou06b, Rou08]. We begin with some definitions and then discuss several properties of patch manipulation.

3.1 Elements of Patch Theory

In this section we make precise terminology that is commonly used in the Darcs community.

Patch Theory is designed to allow users to independently change their data and then share those changes. A patch is a way of recording, storing and communicating changes. Before we give a precise definition of patch we define some of the important concepts in Patch Theory.

Definition 3.1.1. *A* **repository** *consists of a sequence of patches and a working copy.*

The sequence of patches in the repository represents a set of changes. We want the user to work with patches in such a way that the set of changes define the exact contents of the repository and allow the user to think in terms of sharing changes between repositories. For example, we would like for a merge of two repositories to be simply the union of their sets of changes.

Each repository may have several *states*. For example, the state that results from applying all of the patches in the repository is called the *pristine state*.

Definition 3.1.2. *A repository* **state** *is a collection of directories, files and the contents of those files.*

We give a special name to the pristine state as it gives us a convenient way to discuss the effect of applying patches while ignoring any changes that have not yet been recorded by the user.

The working copy of the repository is where users do their work between version control operations. In Darcs the working copy is a directory storing the user's files and data as the user currently chooses to see it and work with it. Example operations involving patches are removing patches, recording new patches or applying patches from a different repository and doing so will result in a new working copy corresponding to a new state.

Definition 3.1.3. *A* **context** *is a sequence of patches that can be applied to the empty state. The empty state refers to an empty collection of directories and files.*

We can now give a more precise definition of patch.

Definition 3.1.4. *A* **patch** *is a concrete representation of a change made to the state of a repository. Each patch is a transformation on repository state, and must be an invertible transformation. Each patch also depends on a* context *as defined in Definition 3.1.3.*

A few example patch types include, change to file contents, renaming a file, as well as file additions and deletions.

We will use bold capital letters (e.g. $\mathbf{A}$, $\mathbf{B}$) to refer to *patches*.

Each patch has exactly two contexts, the context required to apply the patch, the pre-context, and the context that results from applying the patch, the post-context.

Definition 3.1.5. *The* **pre-context** *of a patch is the context that exists prior to the patch and is required to apply the patch. Similarly the* **post-context** *of a patch is the context that results from appending the patch to the pre-context.*

Lowercase italic letters will refer to *contexts*, and will be placed in the superscript position in order to describe the pre- and post-contexts of a patch, as in ${}^{o}\mathbf{A}^{a}$. For example, if the repository has a context of o and the user then edits one file and records a new patch $\mathbf{A}$, then the context might then be a. Thus, the user has created a patch with pre-context o and post-context a. To denote this we would write ${}^{o}\mathbf{A}^{a}$, where a is equal to the context o with patch $\mathbf{A}$ appended to it.

A repository might contain two patches, ${}^{o}\mathbf{A}^{a}$ and ${}^{a}\mathbf{B}^{b}$, in which case we could put them in a sequence and simply write, ${}^{o}\mathbf{A}^{a}\mathbf{B}^{b}$. Note that since the post-context of $\mathbf{A}$ matches the pre-context of $\mathbf{B}$ we only write the context a once. Often the contexts may be understood and are omitted, as in $\mathbf{AB}$.

3.2 Commute

When the result of composing two functions is the same regardless of composition order, the functions are said to be commutative. Since our patches contain a transformation of state, we would like to commute patches. Commutation of patches will give us a natural way to reorder sequences of patches and a way to implement merging of patches. If we have two invertible transformations of state, T_1 and T_2 such that

$$T_1 \circ T_2 = T_2 \circ T_1,$$

then we say that the functions T_1 and T_2 are ***commutative*** functions.

We must note that above, T_1 and T_2 are not patches because we have not associated pre- and post-contexts to them. What we mean is that we have two functions with domains and ranges such that they can be composed either way and the resulting transformation of state is the same.

To construct patches from T_1 and T_2 we associate with each a pre-context. Suppose the patch $\mathbf{A}$ was created from a repository of context o, from the transformation T_1, then let $\mathbf{A}$ have pre-context o and let the resulting post-context be a. That is, we have constructed a patch ${}^{o}\mathbf{A}^{a}$. Similarly, suppose the transformation T_2 is then applied and a patch is created with pre-context a and post-context b, let this patch be ${}^{a}\mathbf{B}^{b}$. So far we have constructed ${}^{o}\mathbf{A}^{a}$ and ${}^{a}\mathbf{B}^{b}$ from T_1 and T_2 in such a way that ${}^{o}\mathbf{A}^{a}$ and ${}^{a}\mathbf{B}^{b}$ are restricted versions of T_1 and T_2. That is, ${}^{o}\mathbf{A}^{a}$ and ${}^{a}\mathbf{B}^{b}$ have the same effect on state but may only be applied or composed in their respective pre- and post-contexts.

By construction, T_1 and T_2 are commutative functions and now we investigate what happens when we commute patches by exploring an example.

3.2.1 Example

To understand the difference between commuting functions and commuting patches, we will work through an example involving file renames and modifications to the contents of those files. This example shows how patches are transformed by commutation.

Suppose we have a repository with two specific files named X and Y. We could then define the following transformations of state, which simply rename the files:

- *rename Y to Z*
- *rename X to Y*
- *rename Z to X*

Suppose also, that we make an edit to X and an edit to Y.

Let us name these transformations in general as follows,

$$\begin{aligned} R(x,y) &= \textit{rename } x \textit{ to } y \\ E_1(x) &= \textit{fixed but arbitrary edit to file } x \\ E_2(x) &= \textit{fixed but arbitrary edit, different from } E_1(x), \textit{ to file } x. \end{aligned}$$

Note that in general $E_1(x)$ and $E_2(x)$ depend on the specific contents of the file x.

Using our files X and Y, we see that $E_1(X)$ and $E_2(Y)$ could be applied to the repository in either order. In other words, both $E_1(X) \circ E_2(Y)$ and $E_2(Y) \circ E_1(X)$ transform the repository in exactly the same way. This follows from the contents of X and Y being independent of each other.

Here we will introduce a new patch notation in this section only to make our example commutes more clear. In later sections we will switch back to our more abstract patch notation. Since each patch corresponds to a transformation of state, say T, with specific pre-context a and post-context b, we will denote this: ${}^{a}[\![T]\!]^{b}$

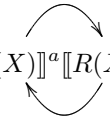
As before we will omit the contexts when it is understood or unimportant. As we will see later, each commute introduces a new pair of patches and this new notation frees us from the task of distinctly naming each patch. This notation also allows us to focus on the state transformation and contexts of the patch.

Our first example uses the patch sequence, ${}^{o}[\![E_1(X)]\!]^{a}[\![R(X,Y)]\!]^{b}$. We are assuming that the context o ensures we have a file X but that no file named Y exists. This sequence of patches edits file X and then renames X to Y.

If we step back and view the above sequence of patches as a composition of transformations, $R(X,Y) \circ E_1(X)$[1], then we see that these transformations are not commutative because it does not make sense to edit the file X after renaming X to Y. The reason is simple, the file X would no longer exist when we try to apply the edit transformation.

Instead of trying to commute $E_1(X)$ with $R(X,Y)$, we could consider $E_1(Y) \circ R(X,Y)$. We arrive at this composition by observing that once X has been renamed to Y we would like to apply our edits to the file Y instead of X. This new composition of transformations would give us the same state as the original composition but with the order of operations reversed. We can apply this idea to swapping the order of patches as well.

Now we swap the order of the patches and reason about the effect on the transformations stored inside the patches,



$${}^{o}[\![E_1(X)]\!]^{a}[\![R(X,Y)]\!]^{b} \rightarrow {}^{o}[\![R(q,r)]\!]^{c}[\![E_1(s)]\!]^{d},$$

[1] The order of function composition is the reverse of the order for patch sequences.

where q, r, and s are placeholders that we will reason about now. A first guess at the values for q, r, and s might be $q = X$, $r = Y$, and $s = X$, but this does not take into consideration the reordering of the operations. When we commute these patches, we must consider whether the transformation $E_1(X)$ affects the transformation $R(X, Y)$. Renaming a file is independent of the contents of the file so we see that the transformation $R(X, Y)$ should not be affected and thus, $q = X$ and $r = Y$. When we consider if $E_1(X)$ is affected by $R(X, Y)$, we realize that the edit should be applied to Y instead of X. After the reordering we are renaming the file before applying the edit, and this means that we must now apply the edit to the new name of the file. Therefore, after the rename of X to Y the edit to X should be applied to Y and we see that $s = Y$. Thus we get the following result,

$$^{o}[\![E_1(X)]\!]^{a}[\![R(X,Y)]\!]^{b} \rightarrow {}^{o}[\![R(X,Y)]\!]^{c}[\![E_1(Y)]\!]^{d}.$$

Finally, notice that the contexts of the patches are different before and after reordering the patches. Context is defined to be a sequence of patches and so reordering the patches changes the sequence. Intuitively, we want the context b to be equivalent to the context d, but we save this discussion for Section 3.2.2.

Now we turn to a slightly bigger example. This time we assume that the context of the repository is such that the files with names X and Y exist but there is no file named Z.

Consider the patches $^{o}[\![E_1(X)]\!]^{a}$ and $^{a}[\![E_2(Y)]\!]^{b}$. Similarly, suppose we create the patch sequence $^{b}[\![R(Y,Z)]\!][\![R(X,Y)]\!][\![R(Z,X)]\!]^{e}$ that swaps the file names of X and Y. For the remainder of this example, we will omit the contexts of the patches, as we are chiefly interested in the effect of commutation on patches. In the following sections we will examine the effect that commute has on context. This gives us a patch sequence,

$$[\![E_1(X)]\!][\![E_2(Y)]\!][\![R(Y,Z)]\!][\![R(X,Y)]\!][\![R(Z,X)]\!].$$

In English, $[\![E_1(X)]\!][\![E_2(Y)]\!]$ modifies the file named X and modifies the file named Y, while $[\![R(Y,Z)]\!][\![R(X,Y)]\!][\![R(Z,X)]\!]$ swaps the names of X and Y. Therefore, this sequence modifies X, modifies Y and finally swaps the file names X and Y.

First we will commute $[\![E_2(Y)]\!]$ all the way to the right and then commute $[\![E_1(X)]\!]$ to the right. When we commute $[\![E_2(Y)]\!]$ with $[\![R(Y,Z)]\!]$ we get $[\![R(Y,Z)]\!][\![E_2(Z)]\!]$, using the same reasoning as the previous example.

Showing this commute as one step we write,

$$[\![E_1(X)]\!][\![E_2(Y)]\!][\![R(Y,Z)]\!][\![R(X,Y)]\!][\![R(Z,X)]\!]$$
$$\rightarrow [\![E_1(X)]\!][\![R(Y,Z)]\!][\![E_2(Z)]\!][\![R(X,Y)]\!][\![R(Z,X)]\!].$$

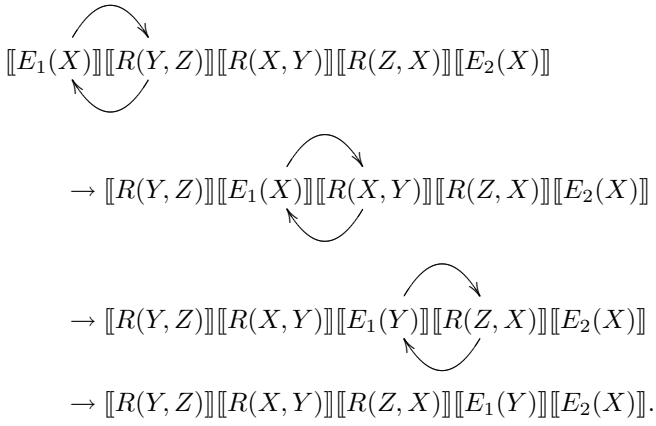
Next we commute $[\![E_2(Z)]\!]$ with $[\![R(X,Y)]\!]$. This time the commute is trivial since the transformations are independent of each other and results in,

$$[\![E_1(X)]\!][\![R(Y,Z)]\!][\![E_2(Z)]\!][\![R(X,Y)]\!][\![R(Z,X)]\!]$$
$$\rightarrow [\![E_1(X)]\!][\![R(Y,Z)]\!][\![R(X,Y)]\!][\![E_2(Z)]\!][\![R(Z,X)]\!].$$

When we commute $[\![E_2(Z)]\!]$ and $[\![R(Z,X)]\!]$ the outcome is similar to the first commute, and we need to update the transformation in the patch $[\![E_2(Z)]\!]$ to modify the file X. The resulting sequence is,

$$[\![E_1(X)]\!][\![R(Y,Z)]\!][\![R(X,Y)]\!][\![E_2(Z)]\!][\![R(Z,X)]\!]$$
$$\rightarrow [\![E_1(X)]\!][\![R(Y,Z)]\!][\![R(X,Y)]\!][\![R(Z,X)]\!][\![E_2(X)]\!].$$

When we commute $[\![E_1(X)]\!]$ through the sequence there is again only two commutes where we update the state transformation. After doing all the commute steps we would have the following sequence,



To summarize, we started from this sequence,

$$[\![E_1(X)]\!][\![E_2(Y)]\!][\![R(Y,Z)]\!][\![R(X,Y)]\!][\![R(Z,X)]\!],$$

and after several commutation steps we arrived at the sequence

$$[\![R(Y,Z)]\!][\![R(X,Y)]\!][\![R(Z,X)]\!][\![E_1(Y)]\!][\![E_2(X)]\!].$$

The two sequences are different operationally but they modify the state of the repository in the same way. In particular, notice that we apply the transformation $E_1(x)$ to Y after the reordering, but before the reordering it was applied to X. The patch containing the transformation $E_2(x)$ underwent a similar modification.

If we had simply treated the state transformations as commutative functions, then we would have an invalid composition of transformations. After all of the reordering in this example $E_1(X)$ would still be a transformation on the contents of a file with name X even though the file with name X was renamed to Y. Thus, $E_1(X)$ would modify the *wrong* file contents.

In the first example we saw that patch commutation always modifies the context of the patches and only *some* of the time changes the state transformation. Also, each new commutation step gives a new sequence yet each sequence defines the same final repository state.

These examples were designed so that all of the patch commutations would succeed, but in general commutation of two patches may not be possible. For example, it does not make sense to commute a patch that creates a file with a patch that modifies that file. We also do not attempt to define patch commute for patches that are not adjacent in a patch sequence.

3.2.2 Abstract Interface

The example in the previous section shows that if we commute state transformations the resulting sequence of transformations is not guaranteed to produce the correct state. Fortunately, the example did illustrate that we can derive new state transformations, and hence new patches, while reordering adjacent patches. This principle is the intuition behind the patch commute operation.

We give the following abstract definition of commute, similar to that found in the Darcs manual [Rou09b].

Definition 3.2.1. *For two patches ${}^{o}\mathbf{A}^{a}$ and ${}^{a}\mathbf{B}^{b}$ we define an operation, which may fail, called* **commute** *such that if ${}^{o}\mathbf{A}\mathbf{B}^{b}$ commutes to the patches ${}^{o}\mathbf{B}_1\mathbf{A}_1^{b}$, then we write ${}^{o}\mathbf{A}\mathbf{B}^{b} \leftrightarrow {}^{o}\mathbf{B}_1\mathbf{A}_1^{b}$.*

While the details of the Darcs commute implementation are beyond the scope of this document, we assume the Darcs patch commute is implemented in such a way that properties such as the following hold.

Property 3.2.1. *Patch commute is self-inverting. For example, if $\mathbf{A}\mathbf{B} \leftrightarrow \mathbf{B}_1\mathbf{A}_1$, then $\mathbf{B}_1\mathbf{A}_1 \leftrightarrow \mathbf{A}\mathbf{B}$.*

Property 3.2.2. *Patch commute preserves the pre-context and gives an* **equivalent** *post-context of the sequence when adjacent patches are commuted. For example, if ${}^{a}\mathbf{A}^{b}\mathbf{B}^{c} \leftrightarrow {}^{x}\mathbf{A}_1^{y}\mathbf{B}_1^{z}$, then it must be the case that $a = x$[2], and we define z to be* equivalent *to c, while the relationship between b and y is unknown. Intuitively we want the sequence that results from commute to define the same repository state.*

We want the above properties so that patch commute will be an *equivalence relation* on sequences of patches. For patch sequences that are related by some number of commutes we write $\leftrightsquigarrow$ and say "can be commuted to." For example, if $\mathbf{A}\mathbf{B} \leftrightarrow \mathbf{A}_1\mathbf{B}_1$, then $\mathbf{A}\mathbf{B} \leftrightsquigarrow \mathbf{B}_1\mathbf{A}_1$ after just one commute.

For the relation $\leftrightsquigarrow$ to be an equivalence relation, it must satisfy the following [Rot02] for all patch sequences x, y and z:

1. $x \leftrightsquigarrow x$;
2. if $x \leftrightsquigarrow y$ then $y \leftrightsquigarrow x$;

[2] This is true because the patch sequence to the left, if any, has not been altered.

3. if $x \leftrightsquigarrow y$ and $y \leftrightsquigarrow z$ then $x \leftrightsquigarrow z$.

Here we take the property that the relation $\leftrightsquigarrow$ forms an equivalence relation for granted much like we assume here that the Darcs commute implementation is correct. That is, the specification of Darcs commute, eg., Patch Theory, specifies that the relation $\leftrightsquigarrow$ must be an equivalence relation and it would be a defect in the Darcs implementation if it were not. For this reason, we do not give a proof here. Providing a rigorous proof that $\leftrightsquigarrow$ forms an equivalence relation is left as future work.

Every equivalence relation partitions elements into disjoint sets known as equivlance classes [Rot02]. Here the equivalence classes are sequences of patches that define the same final repository state, but this is not to say that all sequences that define a common final repository state are in the same equivalence class.

In Definition 3.1.3 we said that a context is a sequence of patches. Now that we can use the relation $\leftrightsquigarrow$ to talk about equivalent sequences of patches we may also talk about equivalent contexts. By equivalent context we mean sequences of patches that are equivalent under the relation $\leftrightsquigarrow$. Equivalent contexts should define identical repository states. To fully define equivalent contexts we also need to consider inverse patches in the next section. Also, we do not distinguish in our notation between contexts that are identical and contexts that are equivalent.

In summary, we see that when it is possible to commute patches, the pre- and post-contexts of the patch sequences are equivalent and the operation of commutation results in *new* patches that are semantically linked to the original patches.

3.3 Inverse Patches

The idea of inverse patches is borrowed from the Darcs manual [Rou09b]. The inverse of patch $\mathbf{B}$ is denoted, $\overline{\mathbf{B}}$, and has the property that the state transformation in $\overline{\mathbf{B}}$ is the inverse of the state transformation in $\mathbf{B}$. We define the pre-context of $\overline{\mathbf{B}}$ to be the same as the post-context of $\mathbf{B}$. The composition $\mathbf{B}\overline{\mathbf{B}}$ results in a context that defines the same repository state as the pre-context of $\mathbf{B}$. For this reason, we define the post-context of $\overline{\mathbf{B}}$ to be equivalent to the pre-context of $\mathbf{B}$. In our notation we write, ${}^{o}\mathbf{B}^{b}$ and ${}^{b}\overline{\mathbf{B}}^{o}$ by the following property.

Property 3.3.1. *Let ${}^{o}\mathbf{B}^{b}$ be a patch and let ${}^{x}\overline{\mathbf{B}}^{y}$ be the inverse patch. We define o to be equivalent to y and b to be equivalent to x.*

The intuition behind this property is that each patch has an inverse patch which nullifies, or undoes, the effects of the patch including resetting to an equivalent context.

3.4 Equality

The properties of patches give rise to the following result which is useful for determining when contexts are equivalent.

Property 3.4.1. *Given two patches, ${}^{x}\mathbf{A}^{y}$ and ${}^{u}\mathbf{B}^{v}$, that contain the same transformation of state, T, then x is equivalent to u, if and only if, y is equivalent to v.*

Property 3.4.1 is useful for proving when contexts are equivalent after performing a series of commutes, or when examining two patches that start or end in the same context.

Note that given two arbitrary patches ${}^{x}\mathbf{A}^{y}$ and ${}^{u}\mathbf{B}^{v}$, Property 3.4.1 does not apply, unless $\mathbf{A}$ and $\mathbf{B}$ share the same transformation. Without this extra condition the states defined by y and v may be the same without the contexts being equivalent.

3.5 Merge

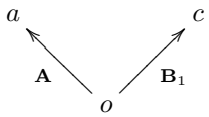
Here we turn to the theory required to merge two sequences of patches. Property 3.5.1 demonstrates how commutation allows us to transform patches by way of commute so that patches that were initially in different contexts may be merged into a sequence. The following property corresponds to Theorem 2 of the Darcs manual [Rou09c].

Property 3.5.1. *Given four patches ${}^{o}\mathbf{A}^{a}$, ${}^{a}\mathbf{B}^{b}$, ${}^{c}\mathbf{A}_1^{b}$, and ${}^{o}\mathbf{B}_1^{c}$ then*

$$ {}^{o}\mathbf{A}^{a}\mathbf{B}^{b} \leftrightarrow {}^{o}\mathbf{B}_1^{c}\mathbf{A}_1^{b}, \text{ if and only if, } {}^{a}\overline{\mathbf{A}}^{o}\mathbf{B}_1^{c} \leftrightarrow {}^{a}\mathbf{B}^{b}\overline{\mathbf{A}}_1^{c}. $$

As we will see later, a valuable property of merge is that it is symmetric. We can see that merge is symmetric by examining how we would use Property 3.5.1 in practice. By this property, we can merge two patches which have the same pre-context and put them into a sequence assuming that they may be commuted.

Using the same patches as the statement of Property 3.5.1, we could visualize the patches as being *parallel*[3]:

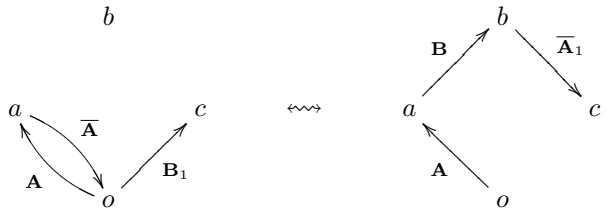


Starting with either ${}^{o}\mathbf{A}^{a}$ or ${}^{o}\mathbf{B}_1^{c}$ we could arrive at two different sequences that share equivalent pre- and post-context. We achieve this with the following steps:

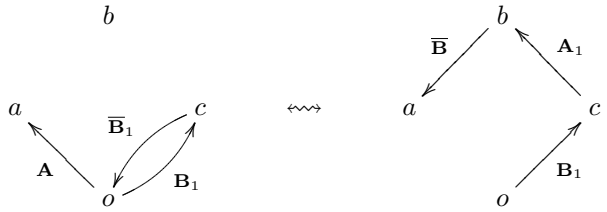
[3]We consider patches that share a pre-context to be parallel whereas patches that share a post-context are said to be anti-parallel.

1. We start by applying the inverse patches ${}^{a}\overline{\mathbf{A}}^{o}$ and ${}^{c}\overline{\mathbf{B}}_1^{o}$ respectively and get ${}^{o}\mathbf{A}^{a}\overline{\mathbf{A}}^{o}$ and ${}^{o}\mathbf{B}_1^{c}\overline{\mathbf{B}}_1^{o}$, corresponding to:

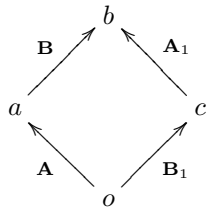
2. Apply ${}^{o}\mathbf{B}_1^{c}$ to the end of the sequence ${}^{o}\mathbf{A}^{a}\overline{\mathbf{A}}^{o}$ and then use Property 3.5.1 to get ${}^{o}\mathbf{A}^{a}\overline{\mathbf{A}}^{o}\mathbf{B}_1 \leftrightsquigarrow {}^{o}\mathbf{A}^{a}\mathbf{B}^{b}\overline{\mathbf{A}}_1^{b}$, corresponding to:



Next, apply ${}^{o}\mathbf{A}^{a}$ to the end of the sequence ${}^{o}\mathbf{B}_1^{c}\overline{\mathbf{B}}_1^{o}$ and again then use Property 3.5.1 to get ${}^{o}\mathbf{B}_1^{c}\overline{\mathbf{B}}_1^{o}\mathbf{A}^{a} \leftrightsquigarrow {}^{o}\mathbf{B}_1^{c}\mathbf{A}_1^{b}\overline{\mathbf{B}}^{a}$, corresponding to:



3. Remove patches ${}^{b}\overline{\mathbf{A}}_1^{c}$ and ${}^{b}\overline{\mathbf{B}}^{a}$ from the right end of their respective sequences. This leaves us with two different sequences of patches having equivalent pre- and post-contexts. By being explicit about the context of the patches, we see that we are left with ${}^{o}\mathbf{A}^{a}\mathbf{B}^{b}$ and ${}^{o}\mathbf{B}_1^{c}\mathbf{A}_1^{b}$. We can also see the symmetry of merge visually:



The symmetry of merge is important because it means patches can be merged in any order and the resulting repository will have an equivalent context which in turn means it will have the same state. The symmetry of merge is what allows us to realize our goal of letting users treat a repository as an unordered collection of changes.

3.6 Summary

The core of Darcs relies on manipulating patches in several key ways:

- There is a commute function that takes two patches and either fails or returns two new patches which correspond to similar transformations of state but have slightly different pre- and post-contexts.
- By commuting patches in sequences we are able to relax the definition of context to equivalent contexts.
- The pre- and post-context of each patch must be carefully tracked to avoid data corruption. This includes contexts which only exist temporarily, or theoretically, as patch sequences are commuted.

As the example commute in Section 3.2.1 shows we cannot just apply patches whenever the state matches the domain of the patch's state transformation. Doing so could lead to different results depending on the order the patches are applied in. To avoid data corruption we use commute when we need to reorder patches. The goal of our work is to make sequence manipulations safe and give static guarantees about that safety. Here, "safe" means that the contexts are always respected and data corruption due to applying patches in the wrong context is avoided.

In the next chapter we will re-examine the properties defined in this chapter to see which ones may be statically enforced by the Haskell type checker.

Chapter 4

Checked Invariants

Now that we have established the most fundamental properties and constraints of Darcs patch manipulation in Chapter 3, we will show our way of encoding the invariants into Haskell types.

Our goal is to ensure the properties from Chapter 3 are checked at compile time. We also seek to find a balance between spending all of Darcs development time on correctness versus writing new code and adding useful features. An overview of the properties we cover is given in Table 4.1.

Table 4.1: Patch Theoretic Properties

Property	Description	Discussed in Section
Definition 3.1.4	Patch	Section 4.5
Definition 3.1.5	Pre- and post-context	Section 4.5
Definition 3.2.1	Commute	Section 4.7
Definition 3.1.1	Repository and patch sequence	Section 4.8
Property 3.5.1	Merge	Section 4.9
Property 3.4.1	Patch equality	Section 4.10

Throughout this chapter we make use of existentially quantified types and Generalized Algebraic Data Types (GADTs). A brief introduction to existential types is given in Appendix A. A brief introduction to GADTs is given in Appendix B.

4.1 Sealed Types

One technique that we rely on heavily is the use of existentially quantified types. We use existentially quantified type variables in several different ways. The most basic appears in our `Sealed` data type.

Existentially quantified types give us a way to mark some of our types as distinct from all other types. We use a special data type, called `Sealed`, to hold the existentially bound types. Using the GADT extension it is defined as follows:

```
data Sealed a where
    Sealed :: a x → Sealed a
```

Using the `Sealed` data constructor the type parameter `x` is hidden inside the `Sealed` type. The only thing we can currently recover about the existentially quantified type `x` is that it exists. This means that when we pattern match on a value of type `Sealed`:

```
f :: Sealed a → ()
f (Sealed a) = ()
```

The type system must invent a new type for `x`, referred to as an *eigenvariable,* inside the pattern match of `f`. Again, this eigenvariable for `x` is distinct. The only type it is equal to is itself. We also cannot expose the eigenvariable to a higher level. Although we can pass the eigenvariable to a polymorphic function.

4.2 Witness Types

We consider a witness type to be a type that demonstrates that a particular property is true. The witness acts as evidence of the property.

We use this idea to represent a proof of type equality. The following `EqCheck` type represents an equality check between two types, `a` and `b`, it is written in the GADT notation, explained in Appendix B:

```
data EqCheck a b where
    IsEq :: EqCheck a a
    NotEq :: EqCheck a b
```

If the types `a` and `b` are equal, then we may use the data constructor `IsEq`, otherwise we must use `NotEq`. At the end of the next section we give an example of how this type can witness a proof.

4.3 Phantom Types

A phantom type is a type that has no value associated with it, such as `phantom` in the following:

```
data P phantom = P Int
```

Above, the type variable `phantom` has no value associated with it on the right-hand side of the equal sign. This means that whenever we construct a value of type `P` we may also give a type for `phantom`. Since `phantom` has no value associated with it, it is free to unify with anything in the type system.

For example each of the following is valid, even within the same program:

```
P 5 :: P String
P 5 :: P [Int]
P 5 :: P (IO ())
```

We could imagine each of the above examples as branding the value `P 5`. In other words, one application of phantom types is that they allow us to embed extra bits of information in our types. In particular we want to attach evidence, or proofs, to our types. Which is to say, we want to associate the phantom type with a witness type.

4.4 Example

We would like to combine witness types and phantom types so that our proof carrying types appear as phantom types. A partial justification for this is: *a*) using phantom types allows for our incremental approach discussed in Section 5.1, and; *b*) associating a full patch sequence with each context type would result in an intolerable run-time overhead.

Relying on the accuracy of witness types when they appear as phantom types can be problematic; when a value is constructed the type of a phantom is essentially arbitrary. To be able to rely on the information embedded in phantom types we need ways to control the type unification.

One approach is to hide the data constructors and only expose specialized functions for constructing the datatype. These constructors are often known informally as *smart constructors*. In our case, we might create the `mkIntP` smart constructor which only allows for the construction of values having the type `P Int`:

```
mkIntP :: Int → P Int
mkIntP n = P n
```

We could make a similar smart constructor for values of type `P String`:

```
mkStringP :: String → P String
mkStringP s = P (length s)
```

This works well as long as set of tags is either completely open or closed to a small set of types. The reason is simple, either we provide full access to the data constructor or we make a smart constructor for each allowed tag. Suppose instead that there are specific rules about what is a valid tag but the set of allowed tags is unbounded. Now we need a new approach.

In the previous section we defined the `EqCheck a b` witness type. Now we combine the concept of witness types with existentially quantified types.

As an example, suppose we have another data type `E`, which uses existential quantification on the type variables `a` and `b`:

```
data E where
  E :: a → b → EqCheck a b → E
```

To construct a value of type `E` we must supply three values, a value of type `a`, `b`, and an `EqCheck a b` value. The type of the `E` constructor forms a relationship between the first two input values and the `EqCheck a b` value. To illustrate this point the following is valid:

```
E 1 2 IsEq
```

While, this example is invalid:

```
E '1' 2 IsEq -- invalid
```

The second example could be made valid by using `NotEq` instead as follows:

```
E '1' 2 NotEq -- valid
```

When we pattern match on a value of type `E` we can use the witness type `EqCheck` to gain information about the existentially quantified types `a` and `b`:

```
test :: E → Bool
test (E a b IsEq) = True
test (E a b NotEq) = False
```

At the point of pattern matching in `test` we know more than just which data constructor of `EqCheck` was used, we also recover information about the type equality status of `a` and `b`. In the first pattern match the `IsEq` constructor tells the type checker that `a` and `b` are the same type even though `a` and `b` are existentially quantified. Without this extra information, the type system would treat `a` and `b` as distinct types.

In this example our witness type provides a proof that is stronger than a run-time check. Here, the type checker is able to see that the types are the same. In our example no run-time check is needed and therefore no cast of `a` to `b` is needed, but in some cases it can still be useful. The complications of providing a run-time type equality is discussed in Section 5.2.

When a run-time check is desired to determine type equality we also need a dynamic cast [BS02]. In such a case, a value of type `EqCheck a b` can be useful for passing around the evidence from the equality check. The key point is that by pattern matching on the `IsEq` data constructor we inform the type system that the two types `a` and `b` of the `EqCheck` are equal. This allows us to use the `IsEq` data constructor as a first class proof of type equality at run-time.

4.5 Patch Representation

Before we examine how to represent patch contexts, we first look at how the transformations that make up patches are represented. We begin by looking at a simplified definition of the `Prim` data type in the Darcs implementation. This abstraction is the primitive representation that corresponds most closely to the Patch Theory discussed in Chapter 3.

```
data Prim where
    Move :: FileName → FileName → Prim
    DP :: FileName → DirPatchType → Prim
    FP :: FileName → FilePatchType → Prim
    Identity :: Prim
    ChangePref :: String → String → String → Prim
```

Here we list all of the data constructors that appear in the Darcs source, except the `Split` data constructor which is omitted because it is obsolete. Each one is explained as follows:

- `Move`: Represents a file or directory rename.
- `DP`: The given file name is either added or removed based on the value of `DirPatchType`.
- `FP`: The given file name is either added, removed, or modified based on the value of `FilePatchType`.
- `Identity`: This patch type has no effect on the repository as it represents the identity transformation.
- `ChangePref`: Changes a preference setting for the repository.

The first patch property that we are concerned with is that patches have both pre- and post-context, described in Definition 3.1.5. To encode this in Haskell's type system we use GADTs and phantom types to represent context for patches. Thus we have the following definition:

```
data Prim x y where
    Move :: FileName → FileName → Prim x y
    DP :: FileName → DirPatchType x y → Prim x y
    FP :: FileName → FilePatchType x y → Prim x y
    Identity :: Prim x x
    ChangePref :: String → String → String → Prim x y
```

The phantom types `x` and `y` correspond to pre- and post-context respectively. In our type encoding we are only concerned with contexts that are equivalent, and here we represent only equivalent contexts as described in Section 3.2.2.

One interesting data constructor is `Identity`, which by definition preserves context. Each of the other data constructors corresponds to a type of patch which does change the context and the phantoms express this transformation from `x` to `y`.

The main relationship which is expressed by our use of phantom types is that of how the context is changed by a patch or by a sequence of patches. Although this may seem like a simple relationship, the types that can be expressed this way are still quite helpful in constraining the possible operations and also useful as machine checkable documentation.

4.6 Directed Types

Darcs patches have a notion of transforming between contexts. This naturally leads us to container types that are "directed", and transform from one context to another.

4.6.1 Directed Pairs

The simplest directed type is a directed pair.

```
data (a1 :> a2) x y = forall z. (a1 x z) :> (a2 z y)
data (a1 :< a2) x y = forall z. (a1 z y) :< (a2 x z)
```

Our definition of directed pairs uses a GHC extension that allows type constructors to be infix. We only use infix type constructors because we find them syntactically pleasing. We refer to :> as a forward pair and :< as a reverse pair.

In the above definition the types `a1` and `a2` are the element types in the pair. The `forall` keyword is used to make `z` an existentially quantified type variable. When two types are placed in a forward pair using :> part of each type must match. Suppose we had the two types, `Either String Int` and `Int → Bool`, then we could create the type, `(Either :> (→)) String Bool`. Notice that the `Int` in both types gets hidden due to the existential quantification of `z`. We would need to swap the order of the elements to construct a reverse pair with them as you can tell by looking at where `z` appears in the definition of :<.

Using the `Prim` type we could store a pair of patches:

```
Move "X" "Y" :> Move "Y" "Z" :: (Prim :> Prim) a b
```

Much like our example in Section 4.4, we insert the patches into the forward pair as they are constructed. This acts to partially constrain the phantom types of the `Prim` type and also adds a relationship between the phantom types through the existential quantification in the directed pair.

We use existentially quantified types to represent context for two main reasons, *a*) contexts are implicitly stored by Darcs and, *b*) we need to work with an unbounded number of distinct contexts. Either of the previous two points means we would not be able to manage an explicit type for each context. Thus, we are using the type system to do a great deal of the work for us. We do use one concrete type as a context. We use the Haskell type unit, or `()`, as the type of the empty repository.

4.6.2 Forward Lists

We create the forward list type, which can be used to store types that are parametrized over exactly two other types. One such type is `Prim`, another suitable type for forward lists are functions. For concrete examples using functions see Appendix C.

```
data FL a x z where
    (:>:) :: a x y → FL a y z → FL a x z
    NilFL :: FL a x x
```

In the definition above, `a`, is the element type stored in the list and `x` and `z` are types which enforce an ordering on the elements of the list.

The constructor `NilFL` represents the empty forward list. Because an empty forward list has no elements and carries no transformation we give it the type `FL a x x`.

The constructor `(:>:)`, takes some element with type parameters `x` and `y`, a forward list with the same element type but type parameters `y` and `z`, and produces a forward list with type parameters `x` and `z`. The type `y` is hidden inside the forward list as an existentially quantified type variable. This works for storing elements but it does make some operations tricky as we will see later.

An example of a forward list holding values of type `Prim`:

```
Move "Y" "Z" :>: Move "X" "Y" :>: Move "Z" "X" :>: NilFL :: FL Prim a b
```

The above sequence of patches would swap the names of the files X and Y. Once the list has been constructed if we try to reorder the elements we would get a type error. For example, this function would not be valid:

```
rearrange :: FL Prim x y → FL Prim x y
rearrange (x:>:y) = y:>:x -- This will not type check
```

Once the list is created the context types become fixed. After that we can only put them into a forward list if we respect the relationships between the contexts.

4.7 Expressing Commutation

In Definition 3.2.1, we define commutation of patches as a partial relation. We can now give a type for commute on `Prim` patches:

```
commutePrim :: (Prim :> Prim) x y → Maybe ((Prim :> Prim) x y)
```

The concrete implementation of `commutePrim` is important to Darcs but is not particularly relevant to this discussion and is omitted here. In the actual implementation a type class is used so that commute is polymorphic over the various patch types. Notice that `commutePrim` has a `Maybe` return type. This is because patch commutation is not a total relation. Again, these phantom types represent not a single context, but an entire equivalence class, as described in Section 3.2.2.

4.8 Patch Sequences

Patches have an associated state transformation and we need to apply patches in a way that their contexts are respected. When a patch is recorded we know that it will apply in the current context of the repository. If we also store patches in the order they are recorded, then we know they can also be applied in that order. It would be useful if we had a way to store patches such that their application domains are ensured to be in the correct order.

In Section 4.6.2, we introduce a data type, `FL`, for forward lists. This data type is suitable for storing chains of functions in application order. Here we use forward lists for storing sequences of patches. Instead of storing functions by domain and range, we store patches by pre- and post-context.

Storing patches in context order allows us to bundle up sequences of patches and concern ourselves with just the pre- and post-context of the entire sequence. When extracting elements from the sequence, the context types are lost and we only retain the relationship between context types.

When extracting patches from an `FL` or `RL` sometimes we do know which context a patch should have but our use of existentially quantified types means the type system is pessimistic about context equivalence. To work around this we use patch equality functions described in Section 4.10.

By combining `commutePrim` from the previous section with forward lists we can commute a patch with a sequence of patches. We give this operation the name `commuteFL`:

```
commuteFL :: (Prim :> FL Prim) x y → Maybe ((FL Prim :> Prim) x y)
commuteFL (a :> b :>: bs) = do b' :> a' ← commutePrim (a :> b)
                               bs' :> a'' ← commuteFL (a' :> bs)
                               Just (b' :>: bs' :> a'')
commuteFL (a :> NilFL) = Just (NilFL :> a)
```

The monad instance of `Maybe` handles the cases where `commutePrim` fails and returns `Nothing`. The type checker makes it very difficult now to give an incorrect definition of `commuteFL`.

There are very few incorrect definitions we could give above that would type check. For example, we cannot simply return the input because the type says that the order of the forward list and the patch must be switched in the return value. If we try to return a different list than `b' :>: bs'`, such as `NilFL` or `b' :>: NilFL`, then we will find that the type checker complains.

We could rewrite `commuteFL` so that it returns `a :>: xs :> x`, where `x` is the last element of `bs` and `xs :>: x` is the same sequence as `b :>: bs`. Two other possibilities include returning `undefined` or `Nothing`. Inspecting for one these mistakes is much easier than manually checking that all the steps above respect patch context.

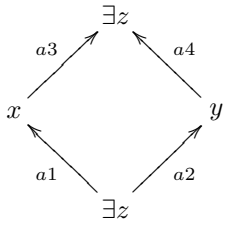
We have been able to implement a full library of sequence manipulations for both forward and reverse lists. Many of the definitions, such as the ones named in Appendix C work on any forward list. Others, such as `commuteFL`, work only for sequences of patches.

4.9 Patch Merge

Property 3.5.1 tells us that when we have two patches which commute and share the same pre-context that we can merge the patches. Whenever patches, or sequences of patches, share a pre-context we say they are parallel. Similarly, when patches, or sequences of patches, share a post-context we say they are anti-parallel. The following types correspond to parallel and anti-parallel pairs:

```
data (a1 :\/: a2) x y = forall z. (a1 z x) :\/: (a2 z y)
data (a3 :/\: a4) x y = forall z. (a3 x z) :/\: (a4 y z)
```

Notice how these definitions correspond to our previous visualization of the symmetry of merge, except that here we are using existential quantification for the pre- and post-contexts of the sequences:



The input to our merge function is a parallel pair, for example for the `Prim` type this would be:

```
merge :: (Prim :\/: Prim) x y → (Prim :/\: Prim) x y
```

The implementation of `merge`, at least for pairs of patches, follows the symmetry of merge example in Section 3.5. Our merge implementation returns the results in an anti-parallel pair because it returns symmetric results. That is, instead of returning just the merged sequence, two patches are returned so that two different sequences, both having the same pre- and post-contexts, can be constructed from the result.

The two sequences that can be built are documented within and constrained by the type signature of `merge`. For example, suppose we have the patches `p1` and `p2` and we use `merge` to get the patches `p1'` and `p2'`:

```
(p1 :\/: p2) :: (Prim :\/: Prim) x y
(p2' :/\: p1') :: (Prim :/\: Prim) x y
```

Using α for the existentially quantified type in the pair `p1 :\/: p2` and β for the existentially quantified type in the pair `p2' :/\: p1'`, the types would be as follows:

```
p1  :: Prim α x
p2  :: Prim α y
p1' :: Prim y β
p2' :: Prim x β
```

The only context preserving sequences we could create with an `FL` are these two:

```
p1 :>: p2' :>: NilFL :: FL Prim α β
p2 :>: p1' :>: NilFL :: FL Prim α β
```

Any other forward list sequences we try to construct from the above four patches would result in type errors!

4.10 Patch Equality

In Section 4.2 we introduced our type witness for type equality functions. Here we use that type to implement parallel and anti-parallel patch equality tests. To implement patch equality we must also introduce an unsafe operation, the problems with this are discussed in Section 5.2. We use the `PatchEq` type class for patch comparison which defines the following functions:

```
class PatchEq p where
    (=\/=) :: p a b → p a c → EqCheck b c
    (=/\=) :: p a c → p b c → EqCheck a b
```

We refer to (=\/=) as parallel equality and (=/\=) as anti-parallel equality. These equality checks for patches are based on Property 3.4.1.

Note that we do require a run-time check to implement both of the above equality functions.

4.11 Summary

Using a combination of phantom, witness and existential types we are able to describe many of the Patch Theory properties in Haskell types. The nature of Haskell's type system means that these properties are checked for us at compile time.

We have not encoded all of the Darcs semantics and there are several key things which we do not express. For example, a more accurate encoding of context equivalence classes would directly use sequences of patches instead of existentially quantified types. We have chosen not to model the context types that way at this time. Partially due to the extra programmer effort, but also because of the extra run-time overhead.

In the next chapter we will discuss these points in more detail as well as the ramifications of applying these ideas to Darcs.

Chapter 5

Discussion

We have outlined the core of the techniques we applied to the Darcs source code. Now we will discuss the implications of working with an existing code base and the direct benefits. The incremental nature of our work is covered in Section 5.1. Many of the pit-falls, setbacks and other hurdles we encountered are covered in Section 5.2. In Section 5.3 we give examples of how this work has improved the Darcs source code.

5.1 Incremental Approach

One of the main challenges with implementing our approach is how to do so in an existing real-world application. Because of this challenge, we needed an implementation plan that is incremental, minimally invasive, and possible without refactoring large parts of the code on the first pass. An incremental approach allows us to work in manageable chunks as volunteer developers can find time. A minimally invasive approach means that there is less risk of introducing new bugs because less code must be changed. Finally, refactoring large parts of the code on the first pass would make it very hard for other developers to review the work.

Following a similar approach to Kiselyov and Shan [KS07], we started in the core of Darcs, where the logic for patch manipulation is defined, to establish a trusted kernel of patch logic. Because our work centers around static guarantees and we have a goal of incremental work, it is natural for us to adopt an implementation strategy that can be enabled or disabled at compile time. We achieved this by using phantom types to carry our witness types giving us the freedom to disable, or remove, the phantom types much like Kahrs [Kah01].

By using techniques that provide compile time guarantees with no run-time component, we were free to incrementally refactor the code base. Even more than working incrementally, we provide two compilation modes giving our approach a distinct feeling of working with a proof assistant. There is the normal compilation mode and there is the witness type, or patch-context aware, compilation mode. The stricter patch-context aware compilation mode is used to verify patch manipulations during development, refactoring, and checking changes from contributors. For creating release binaries, the normal compilation mode is used. The normal compilation mode hides the phantom types, which carry our proofs, from the type system.

To meet our need of having a compile time switch we use C Pre-Processor (CPP) defines. In particular, when compiling the source code with our patch-context aware types we define the CPP symbol `GADT_WITNESSES` and make the following definitions:

```
#ifdef GADT_WITNESSES
#define C(contexts) contexts
#define FORALL(types) forall types.
#else
#define C(contexts)
#define FORALL(types)
#endif
```

The above definitions allow us to write our type signatures as the following example shows:

```
data EqCheck C(a b) where
    IsEq :: EqCheck C(a a)
    NotEq :: EqCheck C(a b)
```

Simply by defining or not defining the symbol `GADT_WITNESSES` we can control at compile time if the code is patch-context aware. The CPP macro `C` surrounds phantom types that represent a context. The CPP macro `FORALL` is useful for mentioning context types when we need to explicitly provide a `forall` in a type signature, for example when using lexically scoped type variables.

Since we are working incrementally using a relatively primitive approach not all of the source code can be compiled when `GADT_WITNESSES` is defined, but this is acceptable for our purposes. Eventually all of the code will be compatible with patch-context types, but for now it allows us to work incrementally while receiving the benefits on the core modules of Darcs where correctness of patch manipulations is of the most importance.

The Darcs development process requires that once all the definitions in a particular module are patch-context aware, that module is added to a special list of modules, named *witnesses*. Whenever patches are accepted to the Darcs source code, the modules in the witnesses list are checked to ensure that each compiles with the macro symbol `GADT_WITNESSES` defined. For this reason, once a module has been converted to be patch-context aware, we lose no safety by compiling Darcs without `GADT_WITNESSES`. In essence, the automated proof assistant is given a chance to reexamine the code whenever it is modified.

A noteworthy, but unplanned, side effect of surrounding context types with the macro `C` is that it helps to visually separate context types from types that are unrelated to context. At first, the macro `C` may seem like visual noise but it is the author's experience that most developers adjust to the notation favorably after using it for a short time.

One minor annoyance with using CPP macros is that the CPP implementation used by GHC cannot handle the single quote character on the same line as a CPP macro. The Darcs source code often uses the single quote character at the end of identifiers to signify an expression which is derived from a previous expression and this can lead to surprising error messages. For example, we modify the forward list append definition to include a single quote character on the same line as a `C` macro:

```
(+>+) :: FL a C(x y) → FL a C(y z) → FL a C(x z)
NilFL +>+ ys = ys'
  where ys' = ys :: FL a C(y z)
(x:>:xs) +>+ ys = x :>: xs +>+ ys
```

Now we get the misleading error message:

```
Not in scope: type constructor or class ‘C’
```

Unfortunately, these errors can be confusing to developers and waste time. The implementation of CPP used by GHC is designed for pre-processing C code and makes assumptions about legal identifier characters. This error happens because the single quote character is a valid identifier character in Haskell identifiers but it is illegal in C identifiers.

5.2 Difficulties

The approach we have taken is not without difficulties and trade-offs. In this section we outline the major problems we encountered.

5.2.1 Intentional Context Coercion

Although not defined in the Haskell 98 report, many Haskell implementations provide a function for arbitrarily changing the type of an expression. This function is commonly given the following name and type signature: `unsafeCoerce :: a → b`

This function, `unsafeCoerce` intentionally circumvents type safety to give the programmer ultimate control over the types in the program. This ability to circumvent type safety puts the burden of type soundness on the programmer, which is occasionally useful.

We apply a restriction to the generality of `unsafeCoerce` so that it can only affect part of the type of a value. We define the following patch coercion function:

```
unsafeCoerceP :: a x y → a b c
unsafeCoerceP = unsafeCoerce
```

There are times when we need to coerce, or change, context explicitly. One reason for this is that our contexts depend on run-time values, but we have other uses for `unsafeCoerceP` which arise from a purely pragmatic standpoint.

The following two sections, 5.2.1 and 5.2.1, give illustrations of when we use context coercion.

Context Equivalence

The development process for Darcs requires that any use of a function having a name that begins with "unsafe" be carefully scrutinized. In practice, the use of `unsafeCoerceP` is not common and the scrutiny happens on the public Darcs mailing list when source changes are submitted by contributors. One goal of Darcs development is to compartmentalize all uses of unsafe functions to a core set of modules that provide safe interfaces.

As an example of compartmentalizing unsafe functions, we favor the use of the type class function `=\/=` over the use of `unsafeCoerceP`. Although `=\/=` is defined in Section 4.10, we give the definition here as well for convenience:

```
class PatchEq p where
    (=\/=) :: p a b → p a c → EqCheck b c
    (=/\=) :: p a c → p b c → EqCheck a b
```

We give an example instance based on a trivial patch type `P`:

```
data P a b = P
instance PatchEq P where
  P =\/= P = unsafeCoerceP IsEq
  P =/\= P = unsafeCoerceP IsEq
```

The instance of `PatchEq` for the `Prim` type is slightly more involved but in essence the instance simply compares the patches for structural equality while relying on the type signature to constrain when the equality check is allowed. We will use this `PatchEq` instance for `P` in the next section as well.

The patch equality checks given here use type witnesses to carry information gained by doing the equality check. The techniques typically used in the literature require that the set of types which can be cast be known fully by the programmer

to avoid the use of `unsafeCoerce`. Instead of using `unsafeCoerce`, a function for dynamic casting is provided between the types. This typically requires creating a type class instance for the types.

This technique can be found in the Haskell library, Typeable [BS02]. Note that, even Typeable can be used to derive unsound functions with type `a → b` by creating "malicious" type class instances [Kis09].

Interfacing with Older Modules

In the Darcs implementation, the `DarcsRepo` module is being phased out in favor of the newer `HashedRepo` module, which uses hashes for improved robustness and atomicity of operations. The `HashedRepo` module is written internally with our witness type style whereas the older `DarcsRepo` module only supports the witness types superficially in most places. Consider the `read_repo` function which reads from a repository and returns the set of patches stored in the repository:

```
read_repo :: RepoPatch p ⇒ Repository p C(r u t) → IO (PatchSet p C(r))
read_repo repo@(Repo r opts rf _)
    | format_has HashedInventory rf = do ps ← HashedRepo.read_repo repo r
                                         return ps
    | otherwise = do Sealed ps ← DarcsRepo.read_repo opts r
                     return $ unsafeCoerceP ps
```

When reading the sequences of patches from the repository, we know that the returned sequence of patches have a final context that matches the recorded context of the repository. The type signature of `read_repo` expresses this through the type `r`. The function `DarcsRepo.read_repo` in the `otherwise` branch of the function does not have the right type to express this relationship whereas `HashedRepo.read_repo` does. The reason for this lack of expressiveness is for entirely pragmatic reasons. To avoid rewriting the older `DarcsRepo` interface we carefully use `unsafeCoerceP` so that the sequence of patches returned by `DarcsRepo.read_repo` will unify with the sequence of patches returned by `HashedRepo.read_repo`.

Examples such as `read_repo` are not common, and can be avoided in theory, but in practice `unsafeCoerceP` can be used to save significant effort when the pay off for that effort is small.

5.2.2 Unsound Equality Examples

While `unsafeCoerceP` has legitimate uses, we must be quite cautious about one particular usage. If we are not careful we can combine =\/= with certain other functions, and completely circumvent the safety of Haskell's type system. While this is very undesirable, we can learn to avoid this by understanding the examples in this section.

If we combine =\/= with a function that returns a type involving phantoms types, and those phantom types are unconstrained with respect to the types of input parameters, then we can recreate `unsafeCoerce :: a → b`.

The following examples demonstrate the problem of recreating an unsafe operation. Note, we use an additional feature of GHC for these examples known as *lexically scoped type variables.* The scope of type variables is introduced by the use of an explicit `forall` in the type signature.[1]

The first example uses a data constructor `P` that allows us to assign arbitrary types to its phantom types. We are using this as a place holder for real patch types. We assume here that =\/= is defined for the type `P a b` such that it always returns `IsEq`, such as the definition in the previous section. We use this to derive an alternative definition of `unsafeCoerce` as follows:

```
unsafeCoerce :: forall a b. a → b
unsafeCoerce x = case a =\/= b of
                     IsEq → x
                     _ → error "a = b, making this impossible"
  where (a, b) = (P, P) :: (P () a, P () b)
```

Below is another example that demonstrates that any function which returns phantoms that are unconstrained by the input types can be used to reconstruct `unsafeCoerce`. We use `zipWithFL` from Appendix C.3 but here we use the type `Maybe` to work around the type checking difficulties. We could have also avoided the use of `Maybe` and used `unsafeCoerceP` but this example demonstrates that unsound code can be written without needing direct access to `unsafeCoerceP`:

```
zipWithFL :: (forall r s u v x y. a r s → b u v → c x y)
          → FL a q z → FL b j k → Maybe (FL c m n)
zipWithFL f (a :>: as) (b :>: bs) =
  case zipWithFL f as bs of
  Nothing → Just (f a b :>: NilFL)
  Just cs → Just (f a b :>: cs)
zipWithFL _ _ _ = Nothing
```

The above code will type check, but notice that the returned type, `Maybe (FL c m n)`, has phantoms `m` and `n` that are unrelated to the input types. This allows the type checker to unify `m` and `n` with any other types. Consider this example of `unsafeCoerce`:

[1] While it could be argued that we should disallow lexically scope type variables to avoid these unsound definitions, the approaches described in this document are significantly easier to express when using lexically scoped type variables. Additionally, it may be possible in some or all cases where lexically scope type variables are used to instead employ clever usage of standard Haskell expressions and functions, such as `asTypeOf`.

```
unsafeCoerce :: forall a b. a → b
unsafeCoerce x = case a =\/= b of
                     IsEq → x
                     _ → error "a = b, making this impossible"
  where a :: FL P () a
        Just a = zipWithFL f (P:>:NilFL) (P:>:NilFL)
        b :: FL P () b
        Just b = zipWithFL f (P:>:NilFL) (P:>:NilFL)
        f _ _ = P -- The way P is constructed does not matter here.
                  -- f only needs to satisfy the type signature of
                  -- zipWithFL.
```

While the examples here may seem contrived, similar examples have occurred naturally during development making this a very real issue we must consider. We give both examples to illustrate that not only can constructors with phantom types be composed with =\/= in unsound ways, but so can any function which returns a value that has unconstrained phantom types. In this regard, we consider such functions unsafe and avoid them when possible.

We now investigate one exception to this rule. When a function returns an unconstrained phantom type as part of a `Sealed` type our program remains sound. This is because a type that has been hidden within the `Sealed` type cannot be passed up or returned to a higher level than it was existentially bound at. For example, if we returned the type `Sealed (FL c m)` from `zipWithFL`, then we cannot use lexically scoped type variables to get at the type of `n` and the above definitions will not work.

Consider this definition of `zipWithFL` that uses `Sealed`:

```
zipWithFL :: (forall r s u v x y. a r s → b u v → c x y)
          → FL a q z → FL b j k → Sealed (FL c m)
zipWithFL f (a :>: as) (b :>: bs) =
 case zipWithFL f as bs of
 Sealed cs → Sealed (f a b :>: cs)
zipWithFL _ _ _ = Sealed NilFL
```

We could try to combine the result of this `zipWithFL` with =\/= but we no longer have access to the type that was previously named `n` and so the examples using lexically scoped type variables to control how it unifies will no longer apply.

We could still try to exploit the type of parameter `m`, and we have the function =/\= that can be used to make the type in the position that `m` occurs in equal. The problem now is that the type where `n` was stored will be a distinct type every time we pattern match on the `Sealed` type. The tricks above fail because we need to take control of two phantoms on the same type in order to make =/\= or =\/= return the desired witness type. We could try constructing one value of type `FL P a c` by pattern patching on the result of `zipWithFL`. We could then use the pattern match to remove the `Sealed` type. Looking at the result of `a =/\= a` we find that the signature of `unsafeCoerce` makes `a` and `b` distinct and this anti-parallel equality test fails to type check.

Finally we should note that the version of `zipWithFL` above using `Sealed` is still not really a pratical definition because the function parameter is too general to be useful for much. For example, `zipWithFL (.)` and `zipWithFL (,)` both fail to type check. We would need a function parameter that is somehow meaningful while not relying on the relationship expressed by our types.

5.2.3 Improving Context

We would like to restrict the type of `unsafeCoerceP` so that it can only be used to tell the compiler when our existentially quantified phantom types should be equivalent. We have not found a practical way to do this. Below we discuss some of the potential workarounds.

Now that we have added witness types for the contexts it may be possible to turn our phantom types into non-phantom types by giving each context a distinct type based on run-time values. Doing so could provide us more precise context equivalence as described below.

For example, if we assign a unique integer to every patch, then each context could be represented by a sequences of integers. Representing context with sequences of integers allows us to match the definition of context exactly. The added precision would come from run-time knowledge of the patch sequence.

Suppose we only dynamically track and check patch contexts as values, then we would only gain guarantees through more testing. Therefore, we would like a way to reflect these values to types. We could achieve this by dynamically mapping the value that represents each context to a distinct type, much like Kiselyov and Shan [KS04]. The advantage of mapping to types is that we keep our static guarantees.

Instead of relying on existential types to create distinct contexts through eigenvariables, we would be able to precisely assign contexts to the newly created patches based on the current state of the repository. Essentially, our test for context equivalence could take into consideration more precise information from run-time values and rely less on the programmer to determine when types should be equivalent.

Unfortunately, mapping run-time values to types comes at a significant run-time cost [McB02]. The mapping itself is costly, but so is the implicit dictionary passing that GHC uses to implement type classes. As McBride explains, the performance hit may be proportional to the structural size of the type. In this proposed scheme we expect the types to be large. Therefore,

we would also need to verify if the run-time overhead of making this change is detrimental in practice. For example, if we implement this value to type mapping in the future, then we may be able to disable it for release builds.

5.2.4 Type Checking

Another major concern was that of which features we could use in a stable way and with stable releases of our chosen compiler, GHC. In this regard we were restricted to already proposed and implemented extensions of Haskell.

A few of the troubles we encountered include:

- Each major release of GHC seems to come with revised type inference rules for GADTs. In particular, much of our type witness code would not compile initially under GHC 6.8. Each major release of GHC seems to be increasingly conservative about inferring types for GADTs, but each release seems to agree when types are properly and sufficiently annotated. The main problem for GHC seems to be sound type inference in the presence of "wobbly-types"[PVWW06].

- As noted by Eaton [Eat06], we also experienced considerable frustration with type errors. While our compiler permitted us to use it as a light-weight theorem prover, the cost was in deciphering type errors. Although thankfully our compiler had a sense of humor and would occasionally admit, "My brain just exploded..." when type checking a tricky case involving existentially quantified types.

- We are also forced to avoid certain syntactic constructs due to the combination of features we are using. For example, the "let" and "where" clauses commonly used for pattern matching is incompatible with data constructors having existential types.

  We can avoid "let" and "where" in Haskell by introducing a local function definition. If the "let" occurs inside the do-notation of a monad, then we can remove the "let" using the following trick:

```
example a = do let x = 5
               return (a,x)
```

  We could equivalently give this definition of `example`:

```
example a = do x ← return 5
               return (a,x)
```

  By using the latter definition we avoid the problematic "let" but the code is less familiar to Haskell programmers.

5.3 Real-World Improvements

A natural question to ask about the work we have done refactoring Darcs is, "Has this work lead to the discovery of bugs in the existing source code?" The answer is, "Yes!" This section highlights several of the defects and other improvements we discovered as a direct result of refactoring the Darcs source to use witness types. It is important to keep in mind that Darcs has been actively used and developed since 2005 with a test suite containing over 100 test scripts, so many bugs have already been discovered and fixed. Finding new bugs in the patch manipulations is not an easy task. We include code coverage statistics generated by the Haskell Program Coverage (HPC) toolkit [GR07] in Appendix D.

Each of the following sections outlines a benefit we achieved with a concrete example.

5.3.1 Detection of Invalid Patch Sequence Manipulations

A prime example of where the witness types have ensured proper sequence manipulations arose naturally while refactoring the source of the interactive command for listing the change history of a repository.

The command line of Darcs supports an interactive view of changes. The user is able to filter and select patches in the repository then step through the changes in each patch. The format for the changes is the same format Darcs uses to store and transfer patches, so a change that modifies a file would include line numbers and the added or removed lines for the file.

The defect we found was that the filters were applied to the patch sequence as if it were an ordinary list of patches. This has the unfortunate side effect that when the changes are displayed adjacent changes may not be in adjacent contexts. If the patches to be removed were commuted out of the sequence instead of removing patches from the sequence by filtering, then all the remaining patches would have adjacent contexts.

One example where this bug might produce confusing output for the user is when multiple patches in the history modify parts of a file but one or more of the patches is removed due to filtering. The line numbers displayed to the user could be very misleading. While most users might ignore the exact line numbers in the output, interactions with other patch types, such as file renames, could lead to serious confusion.

Given the nature of this bug, specifically that it does not cause any sort of machine detectable corruption and it requires hard to craft examples, means that it is very unlikely we would have discovered it through testing. Due to our witness type refactor we were able to discover this defect before a single user reported it.

5.3.2 Safe and General Functions

As an example of safe and general functions, we focus on `get_choices` and the group of functions that it generalizes.

Originally the Darcs source code had four different functions for separating a sequence of patches based on a user's choices. For example, after the patches are tagged we might want to separate the tagged patches at the start of the sequence from the others. One way of separating the sequence is the following function:

```
separate_first_from_middle_last :: Patchy p ⇒ PatchChoices p
                                → ([TaggedPatch p], [TaggedPatch p])
```

There are three other functions like `separate_first_from_middle_last`, that have the exact same type. Not only was it confusing to have four nearly identical functions but because they have the same types, Haskell's type system could not help the programmer by catching accidental use of the wrong variant.

After implementing directed types, we were able to change to the following type, that uses the type :> to express the relationship between patch contexts:

```
separate_first_from_middle_last :: Patchy p ⇒ PatchChoices p x z
                                → (FL (TaggedPatch p) :> FL (TaggedPatch p)) x z
```

Our directed types further inspired our confidence in type safe refactoring and lead us to a unified interface:

```
get_choices :: Patchy p ⇒ PatchChoices p x y
            → (FL (TaggedPatch p) :>
               FL (TaggedPatch p) :>
               FL (TaggedPatch p)) x y
```

Using `get_choices` we are able to safely partition the tagged patches and allow the types to both constrain and document which portion is first, middle, and last. Using pattern matching the caller of `get_choices` can pick which part of the tagged patches to focus on. Making this change, reduced the amount of redundant code while providing a simpler interface for programmers working with this code.

5.3.3 Detection of Defective Functions

Not all of the functions in the Darcs source code could be converted to use our witness types. One such example was the `rempatch` function, which happened to exist outside of the trusted kernel of patch sequence manipulation code.

```
rempatch :: RepoPatch p ⇒ Named p → PatchSet p → PatchSet p
```

The problem with `rempatch` was that it removed its first parameter from a `PatchSet`. While refactoring the module that contained `rempatch` it quickly became apparent that `rempatch` did not have a valid type in terms of context manipulation. Fortunately, `rempatch` was no longer used anywhere in the source code so it was removed.

While it might seem as though having an unused function is not a big concern, in this case if `rempatch` had been used then repository history corruption would have resulted. Thus, finding and removing this defective function, and others like it, is extremely desirable.

5.3.4 Identification of Redundant Functions

As the number of lines of code grows in software projects it seems that there is a tendency for developers to unintentionally reimplement existing functionality. One major problem with this trend is that it becomes harder to make certain changes as all the redundant implementations may need to be found and changed.

One of the hurdles to eliminating redundant functions from the Darcs source code was that of determining exactly which patch manipulations were identical and which ones were merely similar. By exposing patch contexts at the type level the types serve as a form of documentation. A benefit of the extra documentation is the ability to more easily identify identical patch manipulations. One such example was a function `commute_by`, which was defined outside of the trusted kernel of patch manipulations.

Finding `commute_by` was easy, but there was a lack of confidence about which variation of `commute` it represented. For example, the input parameters may have been in a swapped order compared to the standard `commuteFL` function that it was most similar to. After adding context types in the module where `commute_by` was defined, it was very obvious that `commute_by` did indeed reverse the order of the input parameters compared to `commuteFL`. Not only did the context types make us aware that it was similar to `commuteFL` but having them also made it safe to swap the parameters to `commute_by` in all the places it was called. If we had not swapped the parameters correctly, then type checking would have failed.

5.3.5 Writing New Code is Safer

An obvious expectation of improved type safety is that new code we write will contain fewer bugs. Approximately half of the type witness changes were implemented before Darcs 2.0 was completed. We found that writing new modules from scratch to use our type witnesses was easier, safer and less error prone. In fact, Camp [Lyn09], another Haskell version control system based on Patch Theory, is now being written to use our witness types from the very beginning.

The Darcs mailing list, where developers share their contributions, now contains discussions where developers have attempted to write new code or modify existing code and discovered that the context types do not type check. This usually results in the developer gaining a deeper understand of how to correctly modify Darcs as well as preventing potential bugs [Pet08, Sit08, Sch08].

Chapter 6

Conclusion

We have shown that by combining advanced features of Haskell in clever ways it is possible to use the static type checker of GHC to make Darcs more robust and safer to modify. Our approach also leads to source code which is better documented.

The techniques we use are not common or mainstream practice within the Haskell community and for this reason require training. This is a potential drawback considering that Haskell itself is not a mainstream language to begin with. In practice this may not be as bad as it sounds. The majority of the Darcs source code which uses the techniques discussed here, is contained in the inner most core of Darcs. Only contributors who work on the inner workings of Darcs need to fully understand these techniques. In practice developers who work on the core of Darcs are few already and training them to use our techniques has not been a problem.

The drawbacks to our approach seem acceptable in light of the advantages. For example, several defects were uncovered while changing the code to use our techniques. Our approach also continues to prevent specific classes of new defects from entering the code base while serving a complementary role to testing. Our work on Darcs, as an Open Source project, demonstrates that automated theorem proving has real applications in software development.

Finally, our approach could be used in any Haskell program that needs to respect chains of transformations and the manipulation of those transformations.

Bibliography

[Bit09] BitMover. Bitkeeper. on-line, March 2009. http://www.bitkeeper.com/.

[BS02] Arthur I. Baars and Doaitse D. Swierstra. Typing dynamic typing. In *ICFP '02: Proceedings of the seventh ACM SIGPLAN international conference on Functional programming*, volume 37, pages 157–166. ACM Press, September 2002.

[Buc09] Bjorn Buckwalter. Dimensional. on-line, March 2009. http://code.google.com/p/dimensional/.

[Can09] Canonical Ltd. Bazaar. on-line, March 2009. http://bazaar-vcs.org/.

[CH03] James Cheney and Ralf Hinze. Phantom types, 2003.

[CX03] Chiyan Chen and Hongwei Xi. Implementing typeful program transformations. In *PEPM '03: Proceedings of the 2003 ACM SIGPLAN workshop on Partial evaluation and semantics-based program manipulation*, pages 20–28, New York, NY, USA, 2003. ACM.

[Den09] Aaron Denney. Dimensionalized numbers. on-line, March 2009. http://www.haskell.org/haskellwiki/Dimensionalized_numbers.

[Eat06] Frederik Eaton. Statically typed linear algebra in haskell. In *Haskell '06: Proceedings of the 2006 ACM SIGPLAN workshop on Haskell*, pages 120–121, New York, NY, USA, 2006. ACM.

[FI00] Daniel Fridlender and Mia Indrika. Do we need dependent types? *Journal of Functional Programming*, 10:409–415, 2000.

[Fre09] Free Software Foundation. Concurrent versions system. on-line, March 2009. http://www.nongnu.org/cvs/.

[GHC09a] GHC. 6.6. observing code coverage. on-line, March 2009. http://www.haskell.org/ghc/docs/latest/html/users_guide/hpc.html.

[GHC09b] GHC. Ghc manual, section 8.4.4. existentially quantified data constructors. on-line, March 2009. http://haskell.org/ghc/docs/latest/html/users_guide/data-type-extensions.html#existential-quantification.

[GHC09c] GHC. The glasgow haskell compiler. on-line, March 2009. http://haskell.org/ghc.

[GM08] Louis-Julien Guillemette and Stefan Monnier. A type-preserving compiler in haskell. In *ICFP '08: Proceeding of the 13th ACM SIGPLAN international conference on Functional programming*, pages 75–86, New York, NY, USA, 2008. ACM.

[GR07] Andy Gill and Colin Runciman. Haskell program coverage. In *Haskell '07: Proceedings of the ACM SIGPLAN workshop on Haskell workshop*, pages 1–12, New York, NY, USA, 2007. ACM.

[Gre08] Gabor Greif. Thrists: Dominoes of data. on-line, July 2008. http://www.opendylan.org/~gabor/Thrist-draft-2008-07-18.pdf.

[Hal01] Thomas Hallgren. Fun with functional dependencies. In *Proceedings of the Joint CS/CE Winter Meeting*, pages 135–145, Gteborg, Sweden, January 2001. Department of Computing Science, Chalmers.

[Hin01] Ralf Hinze. Manufacturing datatypes. *Journal of Functional Programming*, 11(5):493–524, 2001.

[Jon00] Mark P. Jones. Type classes with functional dependencies. In *ESOP '00: Proceedings of the 9th European Symposium on Programming Languages and Systems*, pages 230–244, London, UK, 2000. Springer-Verlag.

[Kah01] Stefan Kahrs. Red-black trees with types. *J. Funct. Program.*, 11(4):425–432, 2001.

[Kis09] Oleg Kiselyov. Typeable makes haskell98 unsound. on-line, March 2009. http://okmij.org/ftp/Haskell/types.html#unsound-typeable.

[KLS04] Oleg Kiselyov, Ralf Lämmel, and Keean Schupke. Strongly typed heterogeneous collections. In *Haskell '04: Proceedings of the 2004 ACM SIGPLAN workshop on Haskell*, pages 96–107, New York, NY, USA, 2004. ACM.

[KR05] Andrew Kennedy and Claudio V. Russo. Generalized algebraic data types and object-oriented programming. In *OOPSLA '05: Proceedings of the 20th annual ACM SIGPLAN conference on Object oriented programming systems languages and applications*, pages 21–40, New York, NY, USA, 2005. ACM Press.

[KS04] Oleg Kiselyov and Chung-chieh Shan. Functional pearl: implicit configurations–or, type classes reflect the values of types. In *Haskell '04: Proceedings of the 2004 ACM SIGPLAN workshop on Haskell*, pages 33–44, New York, NY, USA, 2004. ACM.

[KS07] Oleg Kiselyov and Chung-chieh Shan. Lightweight static capabilities. *Electron. Notes Theor. Comput. Sci.*, 174(7):79–104, 2007.

[LM99] Daan Leijen and Erik Meijer. Domain specific embedded compilers. In *PLAN '99: Proceedings of the 2nd conference on Domain-specific languages*, pages 109–122, New York, NY, USA, 1999. ACM.

[LO94] Konstantin Läufer and Martin Odersky. Polymorphic type inference and abstract data types. *ACM Trans. Program. Lang. Syst.*, 16(5):1411–1430, 1994.

[Lyn09] Ian Lynagh. Camp. on-line, March 2009. http://projects.haskell.org/camp/.

[McB02] Conor McBride. Faking it simulating dependent types in haskell. *Journal of Functional Programming*, 12(5):375–392, 2002.

[McK06] James McKinna. Why dependent types matter. *SIGPLAN Not.*, 41(1):1–1, 2006.

[Mic09] Microsoft. Visual sourcesafe. on-line, March 2009. http://msdn.microsoft.com/en-us/vstudio/aa700900.aspx.

[Mon09] Monotone. Monotone. on-line, March 2009. http://monotone.ca/.

[Oka99] Chris Okasaki. From fast exponentiation to square matrices: an adventure in types. In *ICFP '99: Proceedings of the fourth ACM SIGPLAN international conference on Functional programming*, pages 28–35, New York, NY, USA, 1999. ACM.

[Per09] Perforce Software. Perforce. on-line, March 2009. http://www.perforce.com/.

[Pet08] Tommy Pettersson. [darcs-users] darcs patch: resolve issue1111: use correct side of return from partitionrl. on-line, October

2008. http://lists.osuosl.org/pipermail/darcs-users/2008-October/014272.html.

[Pey03] Simon Peyton-Jones. *Haskell 98 Language and Libraries: The Revised Report*. Cambridge University Press, May 2003.

[PVWS07] Simon Peyton-Jones, Dimitrios Vytiniotis, Stephanie Weirich, and Mark Shields. Practical type inference for arbitrary-rank types. *J. Funct. Program.*, 17(1):1–82, 2007.

[PVWW06] Simon Peyton-Jones, Dimitrios Vytiniotis, Stephanie Weirich, and Geoffrey Washburn. Simple unification-based type inference for gadts. In *ICFP '06: Proceedings of the eleventh ACM SIGPLAN international conference on Functional programming*, pages 50–61, New York, NY, USA, 2006. ACM Press.

[Rot02] J.J. Rotman. *Advanced modern algebra*. Prentice Hall Upper Saddle River, NJ, 2002.

[Rou06a] David Roundy. Implementing the darcs patch formalism...and verifying it. on-line, February 2006. http://physics.oregonstate.edu/~roundyd/talks/fosdem2006.pdf.

[Rou06b] David Roundy. Verifying the darcs patch code. on-line, November 2006. http://physics.oregonstate.edu/~roundyd/talks/cs_colloquiem.pdf.

[Rou08] David Roundy. Verifying the darcs patch code. on-line, October 2008. http://physics.oregonstate.edu/~roundyd/talks/droundy-08.pdf.

[Rou09a] David Roundy. on-line, March 2009. http://darcs.net/.

[Rou09b] David Roundy. Darcs user manual. on-line, March 2009. http://darcs.net/manual/.

[Rou09c] David Roundy. Theory of patches. on-line, March 2009. http://darcs.net/manual/node9.html.

[Sch08] Benedikt Schmidt. [darcs-users] darcs patch: use read_repo instead of get_unrecorded in changes. on-line, October 2008. http://lists.osuosl.org/pipermail/darcs-users/2008-October/015131.html.

[Sel09] Selenic Consulting. Mercurial. on-line, March 2009. http://www.selenic.com/mercurial/wiki/.

[Sha04] Chung-chieh Shan. Sexy types in action. *SIGPLAN Not.*, 39(5):15–22, 2004.

[She05] Tim Sheard. Putting curry-howard to work. In *Haskell '05: Proceedings of the 2005 ACM SIGPLAN workshop on Haskell*, pages 74–85, New York, NY, USA, 2005. ACM.

[Sit08] Ganesh Sittampalam. [darcs-users] darcs patch: rewrite partitionfl and partitionrl to reduce the numb... on-line, October 2008. http://lists.osuosl.org/pipermail/darcs-users/2008-October/015251.html.

[SS00] Christian Skalka and Scott Smith. Static enforcement of security with types. In *ICFP '00: Proceedings of the fifth ACM SIGPLAN international conference on Functional programming*, pages 34–45, New York, NY, USA, 2000. ACM.

[Sto05] Mark Stosberg. Interview with david roundy of darcs on source control. *OSDir News*, 2005. http://osdir.com/Article2571.phtml.

[SV06] Alexandra Silva and Joost Visser. Strong types for relational databases. In *Haskell '06: Proceedings of the 2006 ACM SIGPLAN workshop on Haskell*, pages 25–36, New York, NY, USA, 2006. ACM.

[Tig09] Tigris. Subversion. on-line, March 2009. http://subversion.tigris.org/.

[Tor09] Linus Torvalds. Git. on-line, March 2009. http://git-scm.com/.

[VWP06] Dimitrios Vytiniotis, Stephanie Weirich, and Simon Peyton-Jones. Boxy types: inference for higher-rank types and impredicativity. In *ICFP '06: Proceedings of the eleventh ACM SIGPLAN international conference on Functional programming*, pages 251–262, New York, NY, USA, 2006. ACM.

[XCC03] Hongwei Xi, Chiyan Chen, and Gang Chen. Guarded recursive datatype constructors. In *POPL '03: Proceedings of the 30th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 224–235, New York, NY, USA, 2003. ACM.

[XP98] Hongwei Xi and Frank Pfenning. Eliminating array bound checking through dependent types. *SIGPLAN Not.*, 33(5):249–257, 1998.

[XS99] Hongwei Xi and Dana Scott. Dependent types in practical programming. In *In Proceedings of ACM SIGPLAN Symposium on Principles of Programming Languages*, pages 214–227. ACM Press, 1999.

Appendix A

Existentially Quantified Types

Existentially quantified types are an extension to Haskell which allows for greater polymorphism and more expressive types [LO94, Kah01, Sha04, PVWS07]. In Haskell no special keyword, other than universal quantification, is needed as explained in the GHC user manual [GHC09b].

The simplest example of existential quantification is below, where the type variable `x` is existentially quantified:

```
data Exists = forall x. Exists x
```

Using the `Exists` data constructor the type `x` is hidden inside the `Exists` type. Using `Exists` we could put different types in a list:

```
[Exists 1, Exists "hello", Exists 'a']
```

The list has type `[Exists]` and the type of each element is hidden in the `Exists` data constructor. Once a value of type `x` has been wrapped inside the `Exists` type we can recover it through pattern matching, but at that point the type system only knows that a valid type `x` once existed in that spot and so it instantiates a new distinct type, or *eigenvariable*, as a place holder for `x`. If we try to return a value with an eigenvariable as the type, then the checker will complain. Thus, once a value has been wrapped in the `Exists` data constructor we can no longer expose it to a higher level of scope. Although we can pass it to polymorphic functions.

For functions that we want to apply to the value stored in the `Exists` type we have the following:

```
mapExists :: (forall x. x → x) → Exists → Exists
mapExists f (Exists x) = Exists (f x)
```

The function passed to `mapExists` may modify only the value stored in the `Exists` type but not the existentially bound type. Given the example above there are very few functions we could pass to `mapExists`, the identity function `id :: a → a` is one such function.

Any function which tries to make use of this existentially quantified type variable will not be allowed. To allow ourselves to manipulate the values inside the `Exists` constructor we can use type classes. By placing type class constraints on the existentially quantified type variable we ensure that certain operations—those defined by the type class—are permitted.

For example, if we would like restrict the types which can be wrapped in the `Exists` data constructor to types that can be shown using the standard `Show` type class, then we would define the following:

```
data Exists = forall x. Show x ⇒ Exists x
```

By combining type class constraints we can do more interesting operations on otherwise arbitrary extensionally quantified types.

Appendix B

Generalized Algebraic Data Types (GADTs)

Generalized Algebraic Data Types (GADTs) [XCC03, PVWW06, CH03] extend the power of standard Haskell data types with while providing a convenient syntax that is similar to the syntax for giving type signatures.

The example of existentially quantified types in Appendix A could have been given in GADT syntax as follows:

```
data Exists where
  Exists :: x → Exists
```

Although it should be noted that the above definition does not make use of the generalized nature of GADTs. In the GADT syntax each data constructor is specified using the same notation that is used to give function type signatures. This allows us to easily and naturally create data constructors with interesting types.

For example consider the following container type, which does take advantage of the generalization provided by GADTs:

```
data Container a where
  IntContainer :: Int → Container Int
  CharContainer :: Char → Container Char
  StringContainer :: String → Container String
```

We can tell by inspection that any value of type `Container a` will hold either `Int`, `Char`, or `String`.

We could write a function such as the following:

```
contents :: Container a → a
contents (IntContainer i) = i
contents (CharContainer c) = c
contents (StringContainer s) = s
```

When have a value of type `Container a` we know the only possible types for `a` are `Int`, `Char` and `String` but do not know which one we have until we examine the value such as with a pattern match or a case-expression. The type system treats the type variable `a` as being any type. In this way, GADTs are similar to type classes, except they are closed and a pattern match allows us to know exactly the type of `a`.

The syntax for GADTs is very flexible and allows us to combine existential quantification and phantom types. For example:

```
data Example a where
  Exists :: Int → x → Example Int
  Phantom :: Int → Example a
```

In the `Phantom` constructor the type variable `a` remains a phantom type, while in the constructor `Exists` the type variable `x` is existentially quantified, and the type variable `a` has the type `Int` associated with it so it is not a phantom type.

Appendix C

Directed Type Examples

The following examples show how directed lists can be used to store functions where the types correspond to the domain and range of the functions.

The examples here use the following data type declarations for forward pairs, reverse pairs, forward lists and reverse lists respectively. Each of following definitions is also discussed in Chapter 4.

We have directed pairs:

```
data (a1 :> a2) x y = forall z. (a1 x z) :> (a2 z y) -- forward pair
data (a1 :< a2) x y = forall z. (a1 z y) :< (a2 x z) -- reverse pair
```

Forward lists:

```
data FL a x z where
    (:>:) :: a x y → FL a y z → FL a x z
    NilFL :: FL a x x
```

Reverse lists:

```
data RL a x z where
   (:<:) :: a y z → RL a x y → RL a x z
   NilRL :: RL a x x
```

We refer to the types above as being directed because of the type relationships expressed in each data constructor.

C.1 Functions

We are specifically interested in storing transformations in our forward lists, so the description here assumes that the element type `a` has a domain and range assosciated with it. For example, consider the Haskell functions `chr :: Int → Char`, `ord :: Char → Int` and `toUpper :: Char → Char`; the first two map between numeric values and characters and the last converts characters to their uppercased version. In Haskell, function types are created with the type constructor →. For example, we could place the `chr` function at the front of a forward list, which would look like this:

```
chr :>: NilFL
```

and have the type, `FL (→) Int Char`. We could continue in this way by adding the function `ord` to the front of the list to get,

```
ord :>: chr :>: NilFL :: FL (→) Char Char.
```

We could imagine writing a function apply with type, `apply :: FL (→) x y → x → y`, with the following definition:

```
apply NilFL x = x
apply (a:>:as) x = apply as (a x)
```

Then, we could take the forward list, `chr :>: toUpper :>: ord :>: NilFL` and apply it to numeric value of the character `'a'`, to find out the character code for `'A'`, as follows:

```
apply (chr :>: toUpper :>: ord :>: NilFL) 97 ⟹ 65
```

If we try to construct an invalid sequence of function applications where the domains and ranges of the functions are not compatible we will get a type error, such as this example from an interactive session with GHC:

```
Prelude Data.Char Darcs.Patch.Ordered> chr :>: ord :>: toUpper :>: NilFL

<interactive>:1:16:
    Couldn't match expected type 'Int' against inferred type 'Char'
      Expected type: Int -> y
      Inferred type: Char -> Char
    In the first argument of '(:>:)', namely 'toUpper'
    In the second argument of '(:>:)', namely 'toUpper :>: NilFL'
```

And a corresponding `rapply`, as follows:

```
rapply :: RL (→) x y → x → y
rapply NilRL x = x
rapply (a:<:as) x = a (rapply as x)
```

Which would be equivalent to the `apply` example as follows:

```
rapply (ord :<: toUpper :<: chr :<: NilRL) 97 ⟹ 65
```

Equivalently, we could define `reverseRL`, that reverses a reverse list by creating the corresponding forward list, and give this alternate definition of `rapply`:

```
reverseRL :: RL a x z → FL a x z
reverseRL xs = r NilFL xs
  where r :: FL a m o → RL a l m → FL a l o
        r ls NilRL = ls
        r ls (a:<:as) = r (a:>:ls) as

rapply :: RL (→) x y → x → y
rapply rl x = apply (reverseRL rl) x
```

C.2 Filtering

The standard Haskell libraries define a filter function for lists with the type, `filter :: (a → Bool) → [a] → [a]`. This filter function returns all the elements of the input list for which the first parameter of filter returns `True`. We sometimes want a similar functon for forward lists, `filterFL`, but what should the type be?

```
filterFL :: (forall x y. p x y → EqCheck x y) → FL p w z → FL p w z
```

The above type requires our `EqCheck` type. Discussed earlier in Section 4.2.

This gives us a way to remove elements from the forward list when the elements behave as the identity transformation on their type parameters, eg. elements of type `p x x`. This gives us a simplified way to ensure the forward list is still valid after elements are removed. More complex rules could be used to remove elements, such as removing sub-sequences with type `FL p x x` in a more general implementation of `filterFL`.

C.3 Zipping

Another interesting case is the Haskell standard library function, `zipWith`, which has the following standard type and definition:

```
zipWith :: (a → b → c) → [a] → [b] → [c]
zipWith z (a:as) (b:bs) = z a b : zipWith z as bs
zipWith _ _ _ = []
```

The standard `zipWith` function applies a user supplied function pairwise to the elements of two lists. The resulting list is only as long as the shorter of the two input lists. For forward lists, we must take into consideration the order of the elements in the forward list. We use the following type and definition for our `zipWithFL`:

```
zipWithFL :: (forall x y a. → p x y → q x y)
          → [a] → FL p w z → FL q w z
zipWithFL f (x:xs) (y :>: ys) = f x y :>: zipWithFL f xs ys
zipWithFL _ _ NilFL = NilFL
zipWithFL _ [] (_:>:_) = bug "zipWithFL called with too short a list"
```

Here we combine a standard Haskell list with the elements of a forward list.

The following function is not one we use in practice, but going over the derivation of the definition is illustrative of the challenges involved in putting forward lists to use.

Imagine if we wanted to define `zipWithFL` so that it operated on two forward lists instead of one list and one forward list. If we ignore for a moment the difficulty of defining the function parameter, then we might try the following incorrect definition:

```
zipWithFL :: (forall r s u v x y. a r s → b u v → c x y)
          → FL a q z → FL b j k → FL c m n
zipWithFL f (a :>: as) (b :>: bs) = f a b :>: zipWithFL f as bs
zipWithFL _ _ _ = NilFL
```

This would almost work, but it turns out that since `NilFL` requires that the type witnesses be the same, eg., `NilFL :: FL a x x`, we get a type error in the second case, because it would require that `n` and `m` be the same type and consequently `q`, `z`, `j` and `k` must all be the same type. One way to express this is to change the last case to check for explicit `NilFL` in each input list.

```
zipWithFL :: (forall r s u v x y. a r s → b u v → c x y)
          → FL a m n → FL b m n → FL c m n
zipWithFL f (a :>: as) (b :>: bs) = f a b :>: zipWithFL f as bs
zipWithFL _ NilFL NilFL = NilFL
zipWithFL _ _ _ = error "zipWithFL: Input lists are not the same length"
```

We add the last case to catch an unwanted input case, and we update the type signature to reflect the relationship of the types `q`, `z`, `j`, `k`, `m` and `n`. The `NilFL` in the second and third parameter will tell the type checker that `m` = `n` for that case, but also relies on the next observation.

In order to tell the type system that `m` and `n` are equal, we need to either use an `EqCheck` or pattern match on `NilFL` for a value that shares type information with the returned value. This is why we update the phantom types of the two input list parameters to be the same as the returned list. Now when we pattern match on `NilFL`, the type system knows `m` = `n` and expects us to return a list in which the phantom types are equal. It also implies that the phantoms must be `m` and `n` instead of some new phantom types.

The above type signature will not type check. If we tried to give the above definition to the type checker, then we would see that there is a problem with applying `zipWithFL` at the tail of each list. When we pattern patch on the left-hand side in the first case the existentially quantified type variable `y` in the definition of the data constructor, `:>:`, is bound to distinct types in each list. The problem is that we now require that both input lists have equal phantoms but the distinct types bound by the existential quantification cannot be equal. To work around this, we would need to use a type equality check, such as `(=\/=)`. In fact, we define a type equality check using this operator in Section 4.10. For now, suppose that we have a function, `(=\/=) :: a r s → a r v → EqCheck s v`, that gives us an `EqCheck` type witness that represents when the types `s` and `v` are equal:

```
zipWithFL :: (forall r s u v x y. a r s → a u v → c x y)
          → FL a m n → FL a m n → FL c m n
zipWithFL f (a :>: as) (b :>: bs) =
  case a =\/= b of
  IsEq → f a b :>: zipWithFL f as bs
  _ → error "zipWithFL: Input lists are not parallel"
zipWithFL _ NilFL NilFL = NilFL
zipWithFL _ _ _ = error "zipWithFL: Input lists are not the same length"
```

Using `(=\/=)` requires that the element types of the input lists are the same and we update our type signature. The above version will finally type check. The above may not be as general as we could have hoped and is also not the best definition if we are most interested in compile time guarantees. We have two very easy ways to make the above function fail at run-time. We could change both errors to a normal value by returning `Nothing` in those cases and switching the return type to `Maybe (FL c m n)`, but this adds very little other than acknowledging the failure cases. We have not bothered to do this as this is not a function that we found useful in practice. Although, deriving it provides a rather colorful example of how our techniques can complicate the definition of traditional list processing functions.

It is also important to note that changing the type of the input function is not enough to avoid the need for `(=\/=)`. For example, this will not type check:

```
zipWithFL :: (forall r s. a r s → a r s → c r s)
          → FL a m n → FL a m n → FL c m n
zipWithFL f (a :>: as) (b :>: bs) = f a b :>: zipWithFL f as bs
zipWithFL _ NilFL NilFL = NilFL
zipWithFL _ _ _ = error "zipWithFL: Input lists are not the same length"
```

We still fail to because the type checker cannot unify the existentially quantified type inside the constructor, `(:>:)`, of the two forward lists.

C.4 Standard Operations

Several standard list manipulations have proven useful for forward and reverse lists. For example, we have defined the following functions for forward and reverse lists, but here only the name and type of our forward list implementation is listed:

```
lengthFL :: FL a x z → Int
mapFL :: (forall w z. a w z → b) → FL a x y → [b]
mapFL_FL :: (forall w y. a w y → b w y) → FL a x z → FL b x z
spanFL :: (forall w y. a w y → Bool) → FL a x z → (FL a :> FL a) x z
foldlFL :: (forall w y. a → b w y → a) → a → FL b x z → a
allFL :: (forall x y. a x y → Bool) → FL a w z → Bool
```

```
splitAtFL :: Int → FL a x z → (FL a :> FL a) x z
(+>+) :: FL a x y → FL a y z → FL a x z -- Corresponds to (++)
nullFL :: FL a x z → Bool
concatFL :: FL (FL a) x z → FL a x z
```

We have two types of map defined for forward lists. One map results in a standard Haskell list type and the other map, `mapFL_FL` is for the case where the resulting list is still a forward list. Missing from the above list are functions where element comparison must be performed.