# DOMAIN-SPECIFIC LANGUAGES FOR AD HOC DATA PROCESSING

A Dissertation

Presented to the Faculty of the Graduate School

of Cornell University

in Partial Fulfillment of the Requirements for the Degree of

Doctor of Philosophy

by

Jonathan DiLorenzo

December 2020

DOMAIN-SPECIFIC LANGUAGES

FOR AD HOC DATA PROCESSING

Jonathan DiLorenzo, Ph.D.

Cornell University 2020

Ad hoc data is everywhere. There is a plethora of data formats in use, which, due to the specificity of their domain, lack the processing tools that we might otherwise take for granted. We call such data formats, and the data that they represent, *ad hoc*. Ad hoc data is usually stored in file systems and can be organized into larger structures that we call *filestores*, collections of files and folders along with the properties between them.

This dissertation aims to support the usage of ad hoc filestores. We design domain-specific languages for processing filestores with increasingly complex requirements, building on previous work on PADS and Forest. These existing systems accept declarative specifications of ad hoc data formats, as single files and filestores respectively. From these specifications, they generate tools for loading, storing, and validating the data.

Unfortunately, Forest does not adequately deal with large filestores, or cost control in general. Nor does it offer support for correctly managing concurrent operations, which are common in file systems. This dissertation offers solutions to these problems.

We first introduce Incremental Forest, a domain-specific language and system that enables incremental processing of filestores. This language offers a new mechanism, a *delay* construct, for explicitly controlling the costs of loading and storing in filestores. Incremental Forest comes with a customizable cost model, which guarantees that a wide class of costs monotonically decrease as delays increase.

Our next system, Transactional Forest, does away with the delays from Incremental Forest, opting to use an entirely new interface language and abstraction, which offer

automatic incrementality. Additionally, Transactional Forest leverages this abstraction, a zipper, to provide simple, provably correct serializable transactions using optimistic concurrency control.

Finally, the Zipper File System goes beyond designing a domain-specific language, targeting the file system itself to provide deeper control with respect to other users. The Zipper File System uses the ideas from Transactional Forest to provide serializable transactions in a zipper-based file system. It also comes with a translation from POSIX that theoretically allows standard applications to be run on our file system, without changes.

Taken together, these systems use domain-specific languages to enable users to efficiently and correctly manage ad hoc filestores in concurrent settings.

# Biographical Sketch

Jonathan DiLorenzo was born in Arlington, Virginia before promptly moving to Sweden with his mother and two brothers. He attended Rönninge Skola for grades 1 through 6, then moved to the recently opened Nytorpsskolan, a spitting distance from his house. He stayed there until 9th grade, with a brief interlude at the Potomac School in Virginia for 8th grade.

He graduated from IT-Gymnasiet Södertörn in 2008 and spent a few months doing cellular biology at AstraZeneca before taking a road trip around the United States. He then attended the University of Virginia, receiving his B.A. in Computer Science in 2013.

Salud, Dinero, y Amor

Y Tiempo para Disfrutarlos

# Acknowledgements

This work would not have come to fruition without my excellent advisor, Nate! Nate has been my ardent supporter throughout my entire PhD. Both in shaping me into a proper researcher and in raising me up when I was down. Thank you so so much for the great times I've had at Cornell.

Without Kathleen Fisher, I don't think my research would exist. Her ideas in PADS and Forest serve as the foundation of my work, but her insight and mentorship are its walls. Kathleen has been like a second advisor to me and for that I am forever grateful.

Beyond professors, I've had three other collaborators in my research: Erin, thank you for introducing me to the horrifying world of SWAT. Richard, you did amazing work on taming SWAT into a functional iForest specification. Katie, you were a total boss, did a bit of everything, and were an absolute pleasure to mentor.

PhDs are scrutinized by a committee and my committee is the best. When I asked Fred to serve on my committee, he warned me that he would read both my thesis and my thesis proposal. I reasoned that if I had to write them anyway, it would be nice to have someone read them and give me insightful and meticulous comments both. And boy, did he come through. Fred, I can't thank you enough for the detailed edits that you suggested I change my thesis with. Like rephrasing sentences to not end with a preposition. As well as vastly more important changes.

Bobby and Dexter are my favorite teachers at Cornell. I sometimes hear their voices when I am correcting or expanding my mental model of the world. In fact, Steffen and I

would view Bobby's lectures to determine right way to teach. It says much about Bobby that we did not know if he wrote with his left hand because he was actually left-handed, or just because it was optimal since his body wouldn't block the board.

I think I've sat in on more of Dexter's courses than the total number of courses I've actually taken (for a grade) at Cornell. He has a way of simplifying and expressing material that cuts right at the heart of what's there. Furthermore, he just knows so much that he can connect anything he teaches to four other fields. Suddenly, I can't possibly forget what he taught since I have so many anchors in my mind.

Cornell Computer Science in general has been an absolutely fantastic environment in which to be a PhD Student. The cozy nature of the department makes it easy to know everyone. Lorenzo always has a smile and the most earnest way of asking how I was doing. Gün always shows up to the social gatherings and has the craziest stories. Joe was the only one who ever shortened my name to Jon, but that's ok because Joe's the best. Also, his Reasoning about Knowledge is enlightening in the truest sense of the word.

When I was visiting, Ross successfully tricked all of the incoming students into believing that he was a first-year graduate student. It lasted for a solid 10 minutes before he couldn't keep it together. I like to think this says everything that you need to know about Ross, but to be clear: It always felt like he was one of us.

I always appreciated Andrew's dry humor and his insights into any Programming Language's problem imaginable. In fact, at least one of the 'future ideas' listed in this dissertation are based on a passing comment by Andrew.

I thought Cornell PL had everything and everyone it needed, then Adrian showed up. His field of research is definitely cool and adds much to the department, but more importantly, Adrian is basically the coolest. He gave me sourdough starter. 10/10 would bake Adrian's sourdough again.

KB might be as far away from my subject as you get, but I still tried to take a class

with her once. It was awesome. Certainly, the subject matter of modeling a world based on incomplete data is fascinating, but KB's way of delivering it was the key. She was also an exceptionally supportive chair and I loved the way she met with all of the students to talk about what their goals were and how she could help them meet those goals.

Beyond the professors, Cornell CS would not have been the same without my peers. I erroneously feel like the dream team (as I have just now decided to call my year) shaped Cornell CS in its image. Andrew, Fabian, and Xiang, my amazing lab mates and friends of many years. Matthew and Laure, my first- and second-year housemates who coincidentally turned out to be interested in the same area and first-year office as me. Edward, who wears his heart on his sleeve and was always eager to bring the department together. Eston, who I always tell the same story about because it is magically only astounding because seems too amazing to be the protagonist.

My main man Rahmtin who sent me the sweetest text that I have ever received at the end of our first-year. And who loooves crème brûlée almost as much as I do. Tobias, a man of many terrible puns and jokes that hover on the edge of acceptability, just as I do. Thodoris, who's eggcellent questions brought us much joy and just as much knowledge.

Beyond the dream team, I would be remiss to leave out my close friends in other cohorts: Hussam, who taught me the right way to listen to Thunderstruck. Jake and Katie, who share so many of my tastes, and are arguably partially responsible for my current relationship! Ethan, who always beautifully leaned in to my humor. Wil, my excellent mentee from my alma mater who adorably thought he would graduate in 3 years. Soumya, who still walks the fine line between jokes and insults. Eric, my confidante and erstwhile lunch partner in the JNF lab from whom I learned a little salsa. Danny, the amazing advocate for mental health and my excellent drinking buddy and friend, with whom I can transition to a deep conversation in an instant. Rachit, the ludicrously precocious, but bright and thoughtful friend that I never expected I would

make from the younger years.

You are all fantastic friends.

And to my regular TGIF crew, we've had many amazing years. I hope that you have enjoyed them as much as I have. They kept me sane. Shir and Kate, carry the torch for me! I believe in you!!!

For me, most of the best times of grad school have been spent at 109 Lake St, my house of five years and my frequent destination the two years before. Eoin and Mark, who lived there before me, were instrumental to my graduate school experience and I remain close with them to this day. Daniel, with whom I moved into Lake St, is the thorniest yet most heart-warming person I know. Sam, who invited me to live at Lake St, has the amazing ability to join in for ridiculous fun times and thirty minutes later say that he is done with a conversation and needs to go do math. Likely one of the cleverest people I know, which makes for an amazing conversation partner and co-hypothesizer-of-things-we-know-little-about. He is also part of the dream team, immensely supportive, and has so many good qualities that I couldn't possibly mention half. Thanks for always being the best, Sam. Flora, who often instigated these ridiculous fun times and is an infinite font of crazy ideas.

Steffen, another dream teamer, who replaced Sam at Lake St with panache. We were not nearly so close before living together as we are now, which can just as easily go the other way. Between his amazing pasta cooking and bread baking, his willingness and ability to discuss the deep details of our work, our shared stubborness, and his inexhaustible energy for social activities, I cannot state enough how grateful I am to have invited him to live at Lake St (and for him).

Amir, who replaced Daniel at Lake St, should probably be a diplomat. He is the most amazingly relaxed person, who nevertheless has the foresight and diplomacy to tackle any house cohesion problem as it shows up. Few are as all-around great guys as Amir.

The excellent guitar work and our shared humor were definitely benefits as well.

Mischa and Aditya, the final additions to Lake St in my time, have quickly become my fast friends. Aditya, we definitely need to have more cheeky cigar nights, even if we have to watch Pride and Prejudice again to make it happen. Mischa, I never knew how amazing plants could be and how much I wanted them in my life. Your patience is unparalleled and I think you might be the only person I know with proper life skills...

I would be remiss in leaving out my closest friends from undergraduate, with whom I still keep up and who have been excellent moral and social support during my PhD. Jeff, my suite mate and first friend at UVa, it's been real. I look forward to many more years of hanging and at least a few crazy trips. Justin, my thespian compadre. Somehow, in spite of hanging around a bunch of PhDs for the last seven years, I know no one who invests themselves as deeply in a litany of fascinating, useful, or obscure topics as you. A man after my own heart and a true friend. Miriam, my constant companion and drinking buddy with whom I can commiserate. I'll pick up some KBS for when I see you next.

Several people deserve special mentions for supporting me throughout this thesis writing process and pandemic. When I wanted to schedule regular Zoom work dates to be productive together, they humored me and leaned into the concept. This thesis would not be done without Sam, Rachit, Eric, Miriam, Jeff, and, of course, Molly.

Molly, my phenomenal girlfriend of five years. You are, and have always been, an endless font of support. Somehow, you are silly enough to appreciate my ludicrous humor, but empathetic enough to understand even my strange brain. I appreciate you and am proud of you.

Last, but not least, I want to thank my family. My brothers, Christopher and Sebastian, who seamlessly transitioned from eternal pains to close friends and confidantes as I grew up, though they assure me (probably correctly) that the former was more on me than on them. Pauline and Charlie, my literal siblings from another mother.

I have seen you grow and mature from children who I was constantly (and almost certainly irrationally) worried about to strong, independent people with whom I can share anything and everything and receive heartfelt empathy and thoughtful advice. I am so proud of you both.

Mormor och morfar, ni har varit här under hela min uppväxt. Innan jag såg mer av världen, förstod jag inte hur viktigt och ovanligt det var. Tack tack tack för att ni alltid trott på mig och stött mig i allt jag gjort. Jag saknar dig mormor.

Åsa and Gunnar, you were my scientific inspirations from an early age. Åsa, I aspire to have your force of will and ability to pursue your dreams. Both of your constant projects are a reminder that everything can improve, and that's worth working for.

Dad and Carol, you always supported me, particularly in my intellectual growth. Dad, who is as stubborn as I am, and one of the first people I argued incessantly with about some irrelevant minutiae. You share my love of knowledge and while we always complained about your long-winded explanations when we were young, I hold them dear in adulthood. Carol, who, as Dad says, won our hearts through her amazing cooking and kept them through her care and support.

Mom, you raised me. Everything that I am, I am because of you. And frankly, I am very happy with who I am.

And finally, Nina, my incredible, brilliant niece of two years. I can't say that you had much to do with this thesis, but you are a constant source of joy and a reminder of what is really important in life.

**Funding.** My research was funded by the National Science Foundation under grants CCF-1253165, CCF-1422046, CNS-1413972, OAC-1642120; by the Office of Naval Research under grant N00014-15-1-2177; and by the Department of Defense under grant DARPA-BAA-16-41 TA2.

*This dissertation was typeset in Bitstream Charter using LaTeX 2$_\varepsilon$.*

# Table of Contents

Chapter 1

# Introduction

*"In the beginning, there was chaos."*

—Hesiod, Theogony

The world is exploding with data. The amount of data being generated each day is exponentially increasing, but data is useless without applications that can effectively use it. Unfortunately, the format of much of this data is ad hoc. Ad hoc formats are usually invented on an as-needed basis, and, most importantly, lack the plethora of processing tools that widespread, standardized formats enjoy. We use the phrase *ad hoc data* to describe data in ad hoc formats. Such data shows up in diverse fields including hydrology, genetics, telecommunications, finance, and health care.

This dissertation is about designing systems and languages for ad hoc data processing. The goal is to minimize the time users need to spend on the minutiae of correctly digesting data into applications and, perhaps, eventually spitting it back out. We want to free up users' time to focus on the salient details of the data and consider the logic behind processing it. So, we design a family of languages that enable users to describe what their data looks like and, in return, get specialized tools, like parsers and deparsers, for manipulating and analyzing that data.

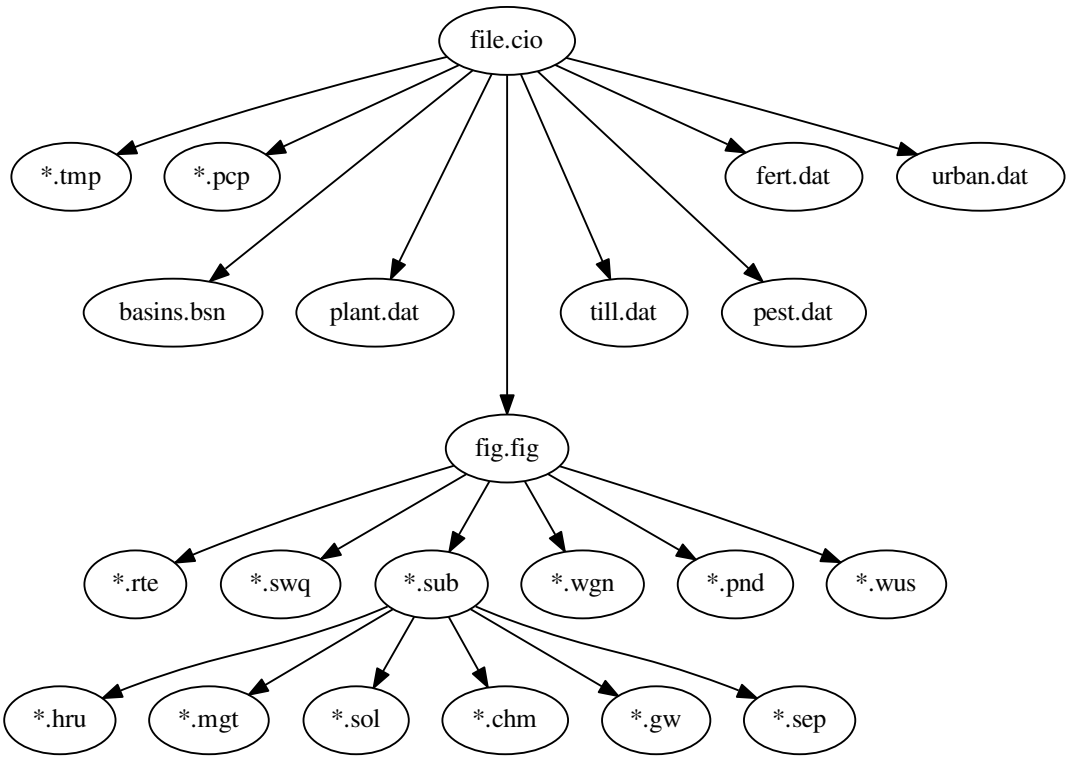
We define ad hoc data not by what it is, but by what it is not: Three data formats that

```
((raccoon:19.19959,bear:6.80041):0.84600,((sea_lion:11.99700, \
seal:12.00300):7.52973,((monkey:100.85930,cat:47.14069):20.59201, \
weasel:18.87953):2.09460):3.87382,dog:25.46154);
```

(a) Newick Standard Data used to represent a philogenetic tree

```
208.196.124.26 - Dbuser [15/Oct/2006:18:46:55 -0700] \
"GET /candatop.html HTTP/1.0" 200 -
www.att.com - - [15/Oct/2006:18:47:01 -0700] \
"GET /images/reddash2.gif HTTP/1.0" 200 237
208.196.124.26 - - [15/Oct/2006:18:47:02 -0700] \
"POST /images/refrun1.gif HTTP/1.0" 200 836
```

(b) Common Log Format (CLF) web server logs




(c) Soil and Water Assessment Tool (SWAT) filestore dependency tree

**Figure 1.1:** Examples of three ad hoc data formats: (a) and (b) are single file formats, while (c) captures the dependency tree of a full filestore, with each individual node being comprised of a different ad hoc format.
*Note:* Backslashes mark newlines that were inserted to improve readability to distinguish them from newlines in the format.

are *not* ad hoc are *XML*, *JSON*, and relational databases. They are structured, standardized formats with built-in tools or easily accessible libraries for processing and querying in every major general-purpose programming language. In comparison, consider the formats in Figure 1.1. The Newick format used in genetics is a flat representation of a tree with weighted edges. The Common Log Format (CLF) is commonly used to log web server activity. Both are single source formats, comprised of single files or perhaps a single source of streaming data. The Soil and Water Assessment Tool (SWAT) [39] Input/Output format is instead an example of a *filestore*, which encompasses a whole collection of files and folders, along with relations between them. In this instance, individual files are additionally in ad hoc data formats (not pictured), but this does not need to be the case. SWAT is used by watershed hydrologists to quantify the impact of hypothetical changes on a watershed.

Note that being ad hoc is an extrinsic property of data. If an ad hoc data format becomes sufficiently standard that processing tools and libraries become widespread, the data format becomes less ad hoc over time. However, it is rare for data formats to grow to become as widely used as *XML* and *JSON*, and thus, it still behooves us to deal with the difficulties that come with processing ad hoc data:

1. The user usually has little or no control over the data format: The data arrives as is and needs to be processed.
2. The documentation is rarely adequate. Frequently, there is no documentation. If it does exist, it tends to be incomplete: The representation of 'missing data' is often undocumented. Additionally, the documentation is often out-of-date due to changing requirements or resource availability. For example, sometimes data that was originally included in a format is never used and it becomes clear that other data would be useful. Instead of changing the format, the old storage locations get repurposed to accommodate the new data.

3. Data may be wrong or malformed. Due to the data being missing, human error, some malfunction, or other issues, some data is different than the expected format suggests. Breaking, discarding the data, or silently failing is rarely a good enough response for processing. The right approach to error handling is application-specific. Errors can even be the most important part of the data: If errors signal that some step in the core business malfunctioned, knowing exactly where this happened is crucial.
4. Formats can feature dependencies between fields. For example, it is common to have one or more *flag* fields indicating the format of the data to come.

While these issues are fairly common in all ad hoc data formats, *filestores*—collections of files and folders and the relationships between them—more frequently suffer from several more:

5. Filestores are often too large to fit into memory. This means that applications cannot load and process all of the data at the same time.
6. File systems are slow. Compounded with the previous point, this leads to different design requirements for applications.
7. File systems tend to support concurrent users. This brings issues that developers must consider to ensure the correctness of their programs.

The standard POSIX file system interface is low-level and requires a large amount of manual effort to deal with the above issues. Further, getting the details right and reasoning about the correctness of programs is famously difficult in low-level languages. The reasoning problem is exacerbated because that the POSIX interface has no formal semantics and, even informally, is underspecified.

Nonetheless, filestores are common. They are often used as a database. There are several reasons why one might use a filestore instead of a database: File systems are

ubiquitous, and the data stored on them is portable. This ubiquity additionally offers a lower barrier to entry. Most users have a file system that they can immediately start using to host their filestore. To use a database, would require: (1) finding appropriate database software, which might have additional costs; (2) transforming raw data into a database compliant format; and (3) learning the interface of that database in the programming language of choice. Additionally, databases tend to be optimized for particular usage patterns and, a priori, users might not know enough about the characteristics of their application to choose a database. If the data is not relational, then finding the right database becomes even harder because non-relational databases tend to be more specialized than relational databases.

Most importantly, as per issue 1 above, many consumers of data are not the producers, so they have no control over what their data looks like.

Whatever a user's reason for using a filestore, these repositories are in common use. These filestores may further contain single-file ad hoc data formats as described above. We view filestores as a general model that can encompass multiple files in multiple, possibly ad hoc, data formats along with higher-level properties of/between files and folders.

The point of this work is to mitigate the issues stated above. We design systems and languages that minimize the application design and programming overhead of ad hoc data processing. In the next section, we concretize the difficulties of working with filestores by considering the requirements of application writers.

## 1.1 Requirements

Consider a watershed hydrologist who wants to run experiments on a SWAT filestore. SWAT, or the Soil and Water Assessment Tool, is a modeling framework used to quantify the impact of hypothetical changes on a watershed. A team of hydrologists might use this

tool to simulate the effects on a river from cutting down a swath of forest and planting crops. What specific steps do the hydrologists need to take to run experiments on this data using their favorite programming language?[1]

First, they need to transfer the data from a filestore into a program. This could be done using string processing techniques and regular expressions, but likely, particularly for such a large format, a hydrologist would be better off writing a lexer and a parser. The hydrologist also needs to write data to the filestore, for use with existing modeling tools. In the case of SWAT, there are many ad hoc data formats for the individual files as well as filestore level dependencies, so many lexers and parsers with additional glue code could be needed.

In order to write a parser, the hydrologists need to know the grammar of the formats in their filestore. In the case of SWAT, there is remarkably complete documentation [38]—650 pages of dense, declarative descriptions of each file format. Translating this documentation into a formal grammar would be non-trivial. Additionally, they need to build robust parsers, which can handle malformed data in application-specific ways. In the case of SWAT, we believe that they would want to be informed of the location of such data in order to manually replace it with something sensible.

Since SWAT filestores can be sizable and many applications do not require reading or writing all of the data, the hydrologists would want to build their parser and deparser to support incremental reading and writing.

Once the hydrologists have built parsers and deparsers with all of these properties, they can finally write their application. Often, these applications take a long time to run, but admit parallelization. For example, calibration often requires many runs that tweak parameters to try to match a model to the ground truth data. The hydrologists parallelize their application for efficiency, but need to ensure that it runs correctly.

[1]There exist a few tools for processing SWAT filestores, but to my knowledge, none have interfaces to general programming languages. The tools are limited in their support for automated and flexible experiments. In the last few years, SWAT+ has been released, which may include a more flexible interface.

Correct concurrency is a difficult problem, made even harder because it is difficult to tell when something went wrong.

Finally, the hydrologists can run their analyses, record the results, and write and publish a paper. This is the exciting part, but notice how many steps are necessary to get here. Importantly, many steps do not require the hydrologists' domain knowledge.

The goal of this research was to mitigate the difficulty of, and time spent on, steps that do not require domain expertise when analyzing data. This work designs systems and languages for automating as many of these steps as possible, with the minimal amount of input from a domain expert. Hopefully, this will free them from worrying about rote details of the problem and enable interesting data processing.

## 1.2 A Solution

We design a family of languages and systems that aid users in ad hoc data processing. The languages allow users to specify the format of single files and full filestores. From these specifications, we generate tools for manipulating data, including incremental parsers and deparsers, and provide automatic serializability between threads running applications against the system's interface. We have formal semantics—for the languages and for a transactional file system that we designed—which lets more advanced users reason about programs.

This line of work started with PADS, a declarative domain-specific language (DSL) for **P**rocessing **A**d hoc **D**ata **S**ources [8]. Users of PADS write a declarative specification of a single ad hoc format. In return, the system generates data types corresponding to that format and generates parsers and deparsers with hooks for application-specific error handling. Additionally, it generates a suite of tools for conversion to XML and some simple statistical analyses.

While PADS deals specifically with single data sources, many ad hoc formats are

collections of directories and files, often themselves in ad hoc formats. Forest [7] is also a DSL, which is similar to PADS, but for specifying and processing whole filestores. Additionally, the developers of Forest observed that these parsers and deparsers are like bidirectional lenses [12], mapping between an in-memory representation and a file system representation of the data. In accordance with this observation, the Forest authors designed a semantics for these mapping functions and proved the standard lens laws: parsing and immediately deparsing should not change the state of the file system; and deparsing, then parsing should return what was originally written.

Forest does not adequately deal with large filestores or allow incremental processing. The authors side-stepped these concerns by using a lazy host language (Haskell). However, it is quite easy for a user's program to attempt to load the entire filestore into memory. For example, if the user tries to check for errors at some level of the data object, everything below that level is loaded. The formal semantics does not reflect the laziness of an implementation, so it can be difficult to predict which operations have this effect and which do not.

Our research on Incremental Forest [4] (iForest, presented in Chapter 3) was designed to deal with these issues. This is a reworking of Forest to allow incremental processing of only those pieces of the filestore required for the particular application in question. The system enables precise and transparent control of costs related to parsing. It features a *delay* construct to specify that a certain subtree of the filestore should only be processed on demand. A tree-transformation language, that we call *skins*, enables users to write a single specification with different delay behaviors and explore tradeoffs between load-granularity and complexity. Additionally, iForest features a modular cost model which admits user-specified notions of cost. As long as the user-provided cost parameters have certain properties, then the cost of parsing decreases monotonically as the number of delays increase. However, iForest does not help users with parallelization

or the issues of concurrency.

We therefore designed Transactional Forest [3] (TxForest, presented in Chapter 5) to provide a simple way for users to manage concurrency and to provide a more parsimonious semantics for incrementality. Transactional Forest retains the specification language of the original Forest, but moves to a novel backend and a new query/processing language. In TxForest, each specification maps to a (tree) zipper, an elegant, functional data structure, which represents a tree in mid-traversal. More specifically, a zipper encapsulates the current node in focus and the path taken from the root to get there. The processing language provides a nice way for users to traverse this structure and to query or update the focus node.

This paradigm automatically enables maximal incrementality, by only parsing or deparsing exactly what is necessary to reach the focus node of the zipper representing the user's filestore. Additionally, the paradigm permits incremental and light-weight backend logging for an optimistic concurrency control scheme that supports serializable transactions. By using TxForest as an interface to their filestore, users automatically get serializable transactions with respect to other TxForest threads. Informally, serializability ensures that the resulting filestore will be as though the TxForest transactions had run in a serial order. This strong semantics makes it significantly easier for users to reason about the correctness of their programs, predicated on the semantics of the file system, which (unfortunately) tend to be informal. The system also cannot give any guarantees with respect to other users of the file system.

The dearth of formal semantics for file systems and our wish to guarantee atomicity in the presence of non-TxForest users motivated us to design the Zipper File System (ZFS, presented in Chapter 6). ZFS is a transactional file system with a zipper as its backend. It is inspired by Kiselyov's work on a zipper file system [27]. ZFS takes a different tack than our other work by requiring users to replace their file system. ZFS is motivated by

our desire to allow concurrency control with respect to arbitrary users. Unfortunately, POSIX file systems offer limited support for mandatory file locking, and such locks can be circumvented [17]. At a similar upfront cost, users can switch over to ZFS to get the stronger semantics.

ZFS provides provably serializable transactions with respect to arbitrary concurrent users. It has a simple, formal semantics with a small interface upon which arbitrarily complex commands (including the POSIX interface) can be built.

Together, ZFS and TxForest (and iForest too), vastly simplify the steps described in Section 1.1, allowing domain experts to expend their efforts on the problems that require domain expertise. In particular, by writing a declarative specification of the filestore and using the TxForest interface, users automatically get parsers and deparsers, which incrementally process their filestore as necessary and ensure serializability between programs running concurrently. This leaves users with only the task of writing application code and running analyses. Additionally, users can go a step further and use ZFS to protect against concurrent execution by other users.

## 1.3 Contributions

In summary, this dissertation makes the following contributions:

- We motivate the need for and present the design and implementation of a domain-specific language and tool for precisely controlling the cost of processing ad hoc filestores (Chapter 3).
- We develop a denotational semantics for a domain-specific language for processing ad hoc data in concurrent settings and prove serializability and round-tripping laws in the style of lenses (Chapter 5).
- We design, formalize, and build a simple transactional file system based on zippers

and provide a translation from POSIX into its semantics (Chapter 6).

## 1.4 Acknowledgments

This dissertation includes contributions that involved coauthors, as described in the following publications and preliminary work:

- Jonathan DiLorenzo, Richard Zhang, Erin Menzies, Kathleen Fisher, and Nate Foster. **Incremental Forest: A DSL for Efficiently Managing Filestores**. In *OOPSLA 2016*. https://doi.org/10.1145/2983990.2984034.
- Jonathan DiLorenzo, Katie Mancini, Kathleen Fisher, and Nate Foster. **TxForest: A DSL for Concurrent Filestores**. In *APLAS 2019*. https://doi.org/10.1007/978-3-030-34175-6_17.
- Jonathan DiLorenzo, Kathleen Fisher, and Nate Foster. **ZFS: A Zipper-based Transactional File System**. Work in progress.

The next chapter introduces a representative example and necessary background information for understanding the rest of this dissertation.

Chapter 2

# Background

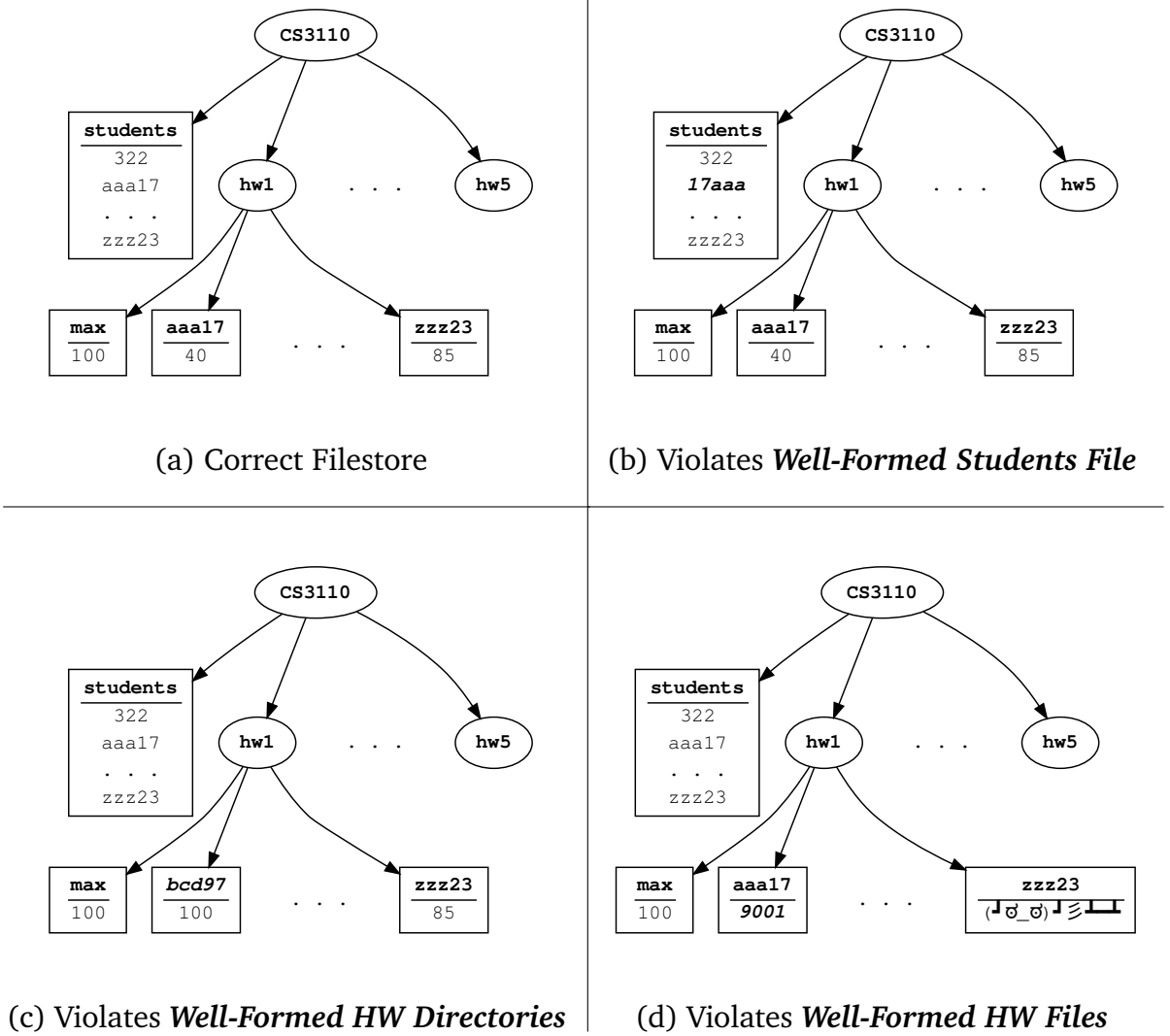
In this chapter, we introduce a running example to illustrate concepts used throughout the dissertation. We design an application for this example looking at POSIX file systems and previous work on PADS [8] (Processing Ad hoc Data Sources) and Forest [7].

For our running example, we use a simplified and idealized course management system. Figure 2.1(a) shows a fragment of a filestore that we could use to track student grades for a course, CS3110: Data Structures and Functional Programming. The top-level directory (**`CS3110`**) contains a file (**`students`**) and a set of sub-directories, one for each homework assignment (**`hw1`**–**`hw5`**). The **`students`** file gives the total number of enrolled students, followed by a list of their NetIDs (Cornell's unique identifier for its students and employees). Each homework directory has a file for each student that contains their grade on the assignment (*e.g.*, **`aaa17`**), as well as a special file (**`max`**) with the maximum possible score. Although this structure is simple, it closely resembles pieces of filestores that have actually been used to keep track of grades at several universities.

There are several key invariants implicit in this structure:

**Definition 2.0.1** (CS3110 Filestore Invariants)**.** The CS3110 filestore is well-formed if and only if the following invariants hold:

1. ***Well-Formed Students File***. The number at the top of the **`students`** file should be equal to the number of lines following it and each of those lines contain only a




(a) Correct Filestore

(b) Violates ***Well-Formed Students File***

(c) Violates ***Well-Formed HW Directories***

(d) Violates ***Well-Formed HW Files***

**Figure 2.1:** Examples of correct and incorrect filestore fragments that store course data.

NetID (*i.e.*, some lowercase letters followed by a number). Figure 2.1(b) violates this invariant.

2. ***Well-Formed HW Directories***. Each homework directory should contain a file named after each of those NetIDs. Figure 2.1(c) violates this invariant.

3. ***Well-Formed HW Files***. For a given student and a given homework, if that student's homework has been graded, then the file with their NetID in that homework directory should contain a number between 0 and the number contained in the **`max`**

file. If that student's homework has *not* been graded, then the file should instead contain a `U`. Figure 2.1(d) violates this invariant.

Though we use these invariants throughout the dissertation, there are other reasonable choices of invariants. For example, perhaps our homeworks can have negative grades. Perhaps we want to distinguish graded and ungraded homeworks in another way, *e.g.* by making the grade of an ungraded homework $-1$. Regardless of the precise choice of invariants, we would like any future application on this filestore to both maintain the invariants and be robust to their breakage.

There are many operations that we want to execute on this filestore—like updating the score of a student on a homework, computing various statistics, and normalizing grades to fit a distribution—but we focus on an operation `addStudent`, which adds a new student to the class. This operation is simple to understand, but captures most issues that are addressed in this dissertation: It exercises every invariant noted above, illustrating the need to capture both within-file and between-file invariants and dependencies. This operation touches every interesting subtree of the filestore, but only a small portion of the nodes, illustrating the need for fine-grained control. Finally, if multiple users try to add a student at the same time, they can run into concurrency errors.

In the remainder of this chapter, we describe how to implement `addStudent` in plain OCaml against a standard POSIX file system, with PADS, and with Forest.

## 2.1 File Systems

This section briefly introduces the POSIX file system interface and examines how to use it to implement the example.

The Portable Operating System Interface (POSIX) defines a standard interface for operating systems. This includes, among other things, a command interpreter with a

standard set of utility programs and a file system interface. We use the term POSIX file system to describe a file system that follows this standard [24].

POSIX file systems are ubiquitous: Most UNIX-based systems use POSIX file systems, and every heavily used general-purpose programming language which we are aware of includes libraries for interfacing with them. These libraries are thus frequently used to interface with filestores, but are not well-suited for this task.

To illustrate this mismatch, we look at an implementation of the `addStudent` example in a general-purpose language using the POSIX interface. We use OCaml as our general-purpose language for the remainder of this dissertation.

```
let addStudent ~student () =
  let (number,students) = get_students () in
  if List.mem students student ~equal:String.equal
  then failwithf "addStudent: Student %s already exists." student ()
  else
    let () = write_students (number+1,student :: students) in
    List.iter (get_hws ()) ~f:(add_student_to_hw student)
```

In this code, `addStudent` takes a string (`student`) as input. This should be the NetID of the student that we wish to add to the course. The operation uses a helper function, `get_students`, to retrieve the current number of students and a list of their NetIDs from the **`students`** file. It checks whether the given `student` is already enrolled in the course, in which case it fails. Otherwise, it proceeds to use another helper function, `write_students`, to update the **`students`** file. Finally, `addStudent` iterates through a list of each homework (obtained through the auxiliary function `get_hws`) adding the new student to each one using `add_student_to_hw`.

This seems simple enough, but most steps were performed by some auxiliary function instead of using the POSIX interface directly. Let us take a look at these functions:

```
let get_hws () =
  Sys.readdir baseDir
  |> Array.to_list
  |> List.filter ~f:(String.is_prefix ~prefix:"hw")

let get_students () : (int * students) =
  let student_file = In_channel.read_lines studentsFilePath in
  let (number,students) = List.split_n student_file 1 in
  number |> List.hd_exn |> int_of_string,students

let write_students ((number,students) : (int * students)) : unit =
  Out_channel.write_lines
    studentsFilePath
    ((string_of_int number) :: students)

let add_student_to_hw student hw =
  Out_channel.write_all (studentPath hw student) ~data:"U"
```

This code utilizes functions that are only one level of abstraction above the POSIX interface. The library function `Sys.readdir` combines POSIX's `opendir`, `readdir`, and `closedir` operations to return a list of the children of the input directory (a path given by a string). The `In_channel` and `Out_channel` modules' suites of reading and writing functions respectively are similarly combining a set of POSIX primitives to provide a convenient user interface. The constants `baseDir` and `studentsFilePath` contain the path to the **`CS3110`** directory and the **`students`** file respectively.

Note the brittleness of this solution. If the filestore looks different from Figure 2.1(a), this code may fail at some unknown point, possibly having already executed a change to the filestore. For example, if there is a file named **`hwstats`** in the base directory, we get this error when trying to add a student with NetID `mqf3`:

```
(Sys_error "code/cs3110/hwstats/mqf3: Not a directory")
```

This is relatively easy to interpret, but the change to the **`students`** file, performed before the failure, persists in the filestore. To resolve this, we need to write significant error- and invariant-checking code. Consider this updated `get_hws` function:

```
let get_hws_err () =
  match Sys.is_directory baseDir with
  | `No -> failwithf "get_hws_err: %s is not a directory" baseDir ()
  | `Unknown -> failwith "get_hws_err: encountered an unknown error"
  | `Yes ->
    let hw_match s =
      let hw_regexp = Str.regexp "hw[0-9]+$" in
      Str.string_match hw_regexp s 0
    in
    Sys.readdir baseDir
    |> Array.to_list
    |> List.filter ~f:hw_match
    |> List.filter ~f:(fun s -> Sys.is_directory_exn (hwPath s))
```

This version requires homework folders to be named **hw***X* where *X* is an integer. Additionally, it checks to make sure that each entry is a folder, skipping them otherwise. We might instead wish to throw an exception to signal that something is wrong with the filestore. In this case, we would also want to prevent the **students** file from being updated, which means that we need to change the main function and possibly others. One option would be to capture a set of changes to be made and only execute them when we know that the union is correct.

One could imagine many other useful checks: We could make sure that a student does not already exist or check that all of the expected invariants hold. We might want to allow other files to coexist with the filestore (like the **hwstats**) or we might wish to explicitly disallow this coexistence by checking that there are no other files and folders. Either way, we need to write a lot of code beyond the core functionality to obtain truly robust code. This is hard enough for a simple filestore like this one, but for more complex filestores with more complex single-file ad hoc data formats, it gets significantly worse. Indeed, just writing the parsing functions *without* any error checking is unmanageable.

The low-level nature of the POSIX interface and the commands derived from it is behind the problems. The interface effectively controls the details of file processing, but capturing the concept and processing of a filestore requires a considerable programming effort.

The low-level nature also makes it difficult to formally (or informally) verify the correctness of application code, an issue exacerbated by the sheer size and complexity of the POSIX interface. POSIX includes around 160 commands [25], all of which are (under)specified in plain English as opposed to a formal language.

In principle, we could formalize the semantics of POSIX, but the low-level nature and complexity make this a tricky proposition. Gardner *et al*. have done work on cutting down the POSIX interface to a more manageable size and providing a formal semantics [15]. They identify 16 core commands from which most others can be built, and they design a separation logic for reasoning about programs written against this interface. However, not all of the complexity of POSIX is captured.

The problems inherent in using the POSIX interface for filestore processing, taken together, lead us to consider other options. Our approach grants users a higher-level abstraction, which should make it easier to write and reason about applications and hopefully disentangle error-handling and core logic.

## 2.2 PADS

One such higher-level abstraction comes from prior work and is called PADS (**P**rocessing **A**d hoc **D**ata **S**ources) [37]. PADS is a declarative domain-specific language and system for processing ad hoc data. With PADS, users specify the shape of their per-file data and, in return, get parsing and printing functions along with a variety of auxiliary tools. This section takes a closer look at PADS and use the running example to illustrate some of its functionality.

Users of PADS write declarative specifications of their data formats, teasing apart their components by giving them names and types and some simple parsing directives, like specifying that a list of similar elements should be separated by newlines and end when the file does. The PADS system then uses this specification to generate format-

specific parsing and printing functions that are robust both to file system and format issues. The parse function, for example, is best-effort, attempting to parse as much of the data as possible and replacing incorrectly formatted values with defaults. Any such error is stored in metadata and is easily available to users as a precise error message.

In the example, both the **students** file format and the grade files in individual homeworks benefit from the use of PADS. In PADS, we might write specifications for these files as follows:[1]

```
pdatatype student =
  | Ungraded of "U"
  | Graded of [i : Pint | $i >= 0 && i <= 100$]

ptype netid = $RE "[a-z]+[0-9]+"$

ptype studentsFile =
  { number : [i : Pint | $i >= 0$]; "\n";
    students : [sl : netid Plist("\n",EOF) | $List.length sl = number$ ]}
```

The `student` datatype describes student grade files. It says that such a file either contains the constant string `U`, in which case it is ungraded, or an integer between 0 and 100, in which case it is graded. The **Graded** description specifies a dependent type: It expects an integer for which the expression on the right of the bar holds. The dollar signs denote antiquotation, allowing users to escape back to the host language, OCaml. Note that this does not take the **max** file into account, though we could do so by manually parsing it and replacing the `100` with the result.

The `studentsFile` type describes the **students** file. It starts with a positive integer, `number`, then a new line, followed by a list of NetIDs, `students`, separated by new lines and terminated at the end of the file (**EOF**). Each NetID is described by the `netid` type, which is a string matching the given regular expression. Additionally, we use dependent types to enforce the invariant that the `number` is equal to the length of the `students` list.

The most obvious benefit of this specification is that it acts as documentation for the

[1]We created this version of PADS, which is available at https://github.com/padsproj/opads

filestore. Ostensibly, it is understandable by humans. In fact, the specification acts as *living* documentation. This documentation will stay up-to-date as the filestore format changes because of a second benefit, which is the generated artifacts: These are (1) types representing the file formats in memory; (2) functions to convert between file system data and these types; and (3) additional tools like statistical aggregators and *XML* and *JSON* converters. Here are a few of the artifacts from the specifications above:

```
type student_rep =
| Ungraded of unit
| Graded of int
type netid_rep = string
type studentsFile_rep = {
  number: int ;
  students: netid_rep list;
}

val student_default_rep : student_rep
val student_default_md : student_md
val student_parse: filepath -> (student_rep * student_md)
val student_to_string : (student_rep * student_md) -> string
val student_manifest : (student_rep * student_md) -> student_manifest
```

The representation types (ending in `rep`) are straightforward translations from the PADS specification, but without the dependencies since these are not expressible in OCaml types. Dependencies are instead captured as values in *metadata*, whose per-specification types (ending in `md`) are also generated, but not shown here. The metadata records errors encountered in parsing, metadata for sub-specifications as well as any extra information that is necessary to reconstruct the original file from the representation.

For each specification, default representations and metadata are generated for use when an error is encountered and as a starting point for construction of new files. The `parse` functions parse the file at their input path according to the function's specification and return a representation and metadata. Conversely, the `to_string` functions use the representation and metadata pairs to construct the contents of the file that they represent. The `manifest` is a natural extension of this concept, preparing to store the in-memory

structures on the file system. The resulting `manifest` type can inform users of potential errors. This is most common in representations that do not match their specification, for example by not meeting the requirements of the PADS dependent types as these are not enforced by the OCaml types. Users can then use a PADS library function (`pads_store`) to store the file described by the manifest at a given path.

We can now rewrite the code to use these artifacts in lieu of providing our own brittle parsers, printers, and error handling:

```
let add_student ~student () =
  let (rep,md) = studentsFile_parse studentsFilePath in
  Pads.exit_on_error md;
  if List.mem rep.students student ~equal:String.equal
  then failwithf "add_student: Student %s already exists." student ()
  else
    let () = add_student_to_studentsFile student (rep,md) in
    List.iter (get_hws ()) ~f:(add_student_to_hw student)
```

We use the generated `studentsFile_parse` function to parse the **`students`** file. Since this function includes error handling, we then use the `exit_on_error` function to exit after printing all errors, should any exist. Otherwise, as before, we make sure that the student is not already enrolled before calling an auxiliary function, `add_student_to_studentsFile`, to add the student to the **`students`** file. Finally, we add the student to each individual homework folder.

As before, the auxiliary functions are doing a some of the heavy lifting. These functions are more complex than their counterparts in the previous section, but they are simultaneously more robust and, in some ways, more understandable:

```
let add_student_to_hw student hw =
  student_manifest (student_default_rep,student_default_md)
  |> Pads.exit_on_mani_error
  |> pads_store (studentPath hw student)
```

```
let add_student_to_studentsFile student (rep,md) =
  let (students, students_md) =
    Pads.insert_into_list
      (student, netid_default_md)
      (rep.students, md.pads_data.students_md)
  in
  let r = {number = rep.number+1; students} in
  let m = {md with pads_data = {md.pads_data with students_md}} in
  studentsFile_manifest (r,m)
  |> Pads.exit_on_mani_error
  |> pads_store studentsFilePath
```

The `add_student_to_hw` function constructs a new ungraded student using the default representation and metadata, creates a manifest, and stores it.

The `add_student_to_studentsFile` function utilizes `insert_into_list`, a PADS library function, to add a new student to both the representation and metadata of the `students` list. This is helpful in making the two line up correctly. We then use the updated lists to construct a new `studentsFile` representation and metadata pair, `(r,m)`, manually updating the `number` of the former. Finally, we construct a manifest and store it back to the file system, exiting and printing any errors if we encounter them.

This approach resolves several of the earlier issues: We no longer need to create our own parsing and printing functions. Additionally, we automatically enforce two of our invariants: ***Well-Formed Students File*** and ***Well-Formed HW Files***. If either is violated, then our metadata will contain an error. Indeed, many other error scenarios are automatically handled. While we choose to print any errors and exit in this example, we could deal with their presence how ever we like, since they are easily accessible and do not cause parsing failures.

However, the previously noted error case, where we add an additional file called **`hwstats`** to the base directory, still causes the exact same issue. Further, we are on our own when it comes to ***Well-Formed HW Directories***. While PADS helps in processing individual files, it does not let us talk about filestore level properties. The next system that we explore attempts to resolve this concern.

## 2.3 Forest

This section introduces prior work on Forest [7], a declarative domain-specific language (DSL) for processing ad hoc filestores. We explore the benefits that Forest can provide using the running example, and look at Forest's syntax and semantics. Additionally, we briefly introduce round-tripping laws in the style of bidirectional lenses [12] and give their instantiation in Forest.

Forest has a higher-level view of an ad hoc data format than PADS, moving away from single-source or single-file formats and instead considering an entire filestore. A filestore is a collection of files and folders that come from the same source, that are important to the same applications, or that represent a single concept. We consider the relationships between the files and folders as part of the filestore: For example, in a Git repository, there is a relation between the record that tracks files under version control and those files.

We can use this broader view of data to capture the entirety of the example format. We would do this by writing a Forest specification like this one:[2]

```
hw = directory {
  max is "max" :: (pads pint);
  students is [student :: (pads student)
               | student <- matches RE "[a-z]+[0-9]+"]}

cs3110 = directory {
  studentList is "students" :: (pads studentsFile);
  hws is [name :: hw | name <- matches RE "hw[0-9]+"]}
```

The `cs3110` specification describes the filestore in its entirety. At the top-level, we have a directory containing two parts: The first is the student list as described by the **`students`** file. The $e :: s$ specification describes a filestore, which, at a path $e$ conforms to specification $s$. In our case, we state that the file **`students`** is described by the PADS

[2]We created this implementation of Forest, which is available at https://github.com/padsproj/oforest

specification `studentsFile`. In the second part, we describe the homework directories as a list of `hw`s, one per path matched by the regular expression `hw[0-9]+`. Each `hw` is a directory, which contains a **`max`** file (containing just an integer), and a list of student files generated by the NetID regular expression.

In these specifications, each directory entry can depend on those preceding it. The current specification does not use this to capture properties like enforcing the max score through the **`max`** file or ensuring that every homework contains a student file if and only if that student is enrolled. However, we could write an alternative specification that does:[3]

```
hw (studentList : studentsFile_rep) = directory {
  max is "max" :: (pads pint);
  students is [student :: (pads student(max))
               | student <- matches RE "[a-z]+[0-9]+"]
              where $check_same_students this_md studentList.pstudents$}

cs3110 = directory {
  studentList is "students" :: (pads studentsFile);
  hws is [name :: hw(studentList) | name <- matches RE "hw[0-9]+"]}
```

In this version, the `hw` description takes a student file (parsed by PADS) as input. It then uses Forest's predicates (written "$s$ `where` $e$" for specifications $s$ and predicates $e$) to assert that the files in the homework directory exactly match the NetIDs of the enrolled students using an auxiliary function, `check_same_students`, that users would be required to write. Additionally, the PADS specification `student` takes an integer as input to determine the maximum allowable score.

Aside from serving as useful living documentation, the Forest system, as with PADS, generates several useful artifacts from this specification. These include tools for filestore visualization, querying, and shell tools along with types and functions like these:

[3]Unfortunately, our implementation of Forest does not support function specifications, but this is a limitation of the implementation rather than of Forest itself.

```
type hw_rep = { max: pint_rep ; students: student_rep list }
type cs3110_rep = { studentList: studentsFile_rep ; hws: hw_rep list }

val cs3110_load: filepath -> (cs3110_rep * cs3110_md)
val cs3110_manifest: ?p:filepath -> (cs3110_rep * cs3110_md) -> manifest
```

The Forest generated types and functions are similar to PADS: There are types for the in-memory representations of each Forest specification. Additionally, there are metadata types (not shown) which store information like errors, sub-metadata, and the path at which the specification is loaded. They also contain various file attributes, like those gathered from the `stat` command in POSIX. The `load` function loads a filestore at an input path and returns its in-memory representation and metadata. The manifest function goes in the opposite direction, optionally taking a path as an argument, which allows users to store their filestore at a different location than where they loaded it from.

Using these functions, we can once more rewrite the code to the following:

```
let add_student ~student () =
  let (rep,md) = cs3110_load baseDir in
  Forest.exit_on_error md;
  if List.mem rep.studentList.pstudents student ~equal:String.equal
  then failwithf "add_student: Student %s already exists." student ()
  else
    add_student_to_filestore student (rep,md)
    |> cs3110_manifest
    |> Forest.exit_on_mani_error
    |> store
```

The first thing to note is that we can now load the entire filestore with a single function. This avoids error-prone auxiliary functions that manually locate the components of the filestore. Secondly, though the Forest `exit_on_error` function essentially looks and acts the same as the PADS version, it is accomplishing more, since the `cs3110` specification captures the entirety of the filestore and its properties, *i.e.* all three of the invariants listed above are checked automatically. Similarly, among other benefits of the Forest `manifest` function, it automatically solves the previous issue where the filestore could end up in an inconsistent state due to an error occuring in the middle of the function. We still rely

on an auxiliary, user-written function, `add_student_to_filestore` to implement much of the functionality however:

```
let add_student_to_studFile student (rep,md) =
  let (pstudents, pstudents_md) =
    Pads.insert_into_list (student, netid_default_md)
      (rep.studentList.pstudents,
       md.data.studentList_md.data.pads_data.pstudents_md)
  in
  let (studentList, studentList_md) =
    set_pstudents_in_studentList ~number:(rep.studentList.number + 1)
      (pstudents, pstudents_md)
      (rep.studentList, md.data.studentList_md)
  in
  set_studentList (studentList, studentList_md) (rep,md)

let add_student_to_every_hw student (rep,md) =
  let add_student_to_hw (hw_rep, hw_md) =
    let path = get_path_exn hw_md in
    let student_md =
      PadsInterface.new_pads_to_forest
        student_default_md (path ^/ student)
    in
    let (students, students_md) =
      Forest.insert_into_comp
        (student_default_rep, student_md)
        (hw_rep.students, hw_md.data.students_md)
    in
    set_students_in_hw (students, students_md) (hw_rep,hw_md)
  in
  map_hws ~f:add_student_to_hw (rep,md)

let add_student_to_filestore student (rep,md) =
  add_student_to_studFile student (rep,md)
  |> add_student_to_every_hw student
```

Instead of storing on their own, auxiliary functions place new data into the existing representation and metadata structures. This lets us leverage type safety to enforce filestore-level properties and centralizes and batches the file system writes. On the other hand, the parts that we wish to add are sometimes deeply nested in the data structure, so they are tedious to add manually. The functions starting with `set_` simplify this process significantly and are straightforward to create (or generate).

We have discussed how the generated artifacts—and even the specification—simplify writing applications against a filestore, *e.g.* by handling errors, invariance checking, and obviating the need for user-written parsing. However, it is not clear how loading and storing functions should behave in the absence of errors. We might have some intuition about how these should work, but translating that understanding to something that the computer understands, or formalizing the correct behavior can be non-trivial. Part of the difficulty arises from the asymmetry between three representations: the specification, the on-disk data, and the in-memory data. The specification captures high-level properties describable by dependent types, while the in-memory structures have no such facility. The on-disk data is essentially untyped, losing even more information. Furthermore, the specification and in-memory representation can quite easily describe filestores that could not exist on a standard file system. Consider the following specification:

```
bad_dir = directory {
  foo1 is "foo" :: file;
  foo2 is "foo" :: directory { ... }
}
```

In a POSIX file system, the same path cannot be both a file and a directory, so this is clearly nonsense (and would indeed signal an error if loaded at any path). We may also have a sensible, albeit odd, specification with a matching in-memory representation that could not possibly exist on-disk:

```
ok_dir = directory {
  foo1 is "foo" :: file;
  foo2 is "foo" :: file
}

let bad_rep = {foo1 = "A string"; foo2 = "A different string"}
```

While a single file could be specified twice in a specification, it cannot simultaneously have two different contents. The core of the issue lies in the difference among the representations: The data of every on-disk filestore can be captured by an in-memory representation, but multiple in-memory representations could map to the same on-disk

$$
\begin{array}{ll}
\text{Paths } p & ::= \cdot \mid p/u \\
\text{Contents } C & ::= \mathsf{File}\ u \mid \mathsf{Link}\ p \mid \mathsf{Dir}\ \ell \\
\text{File Systems } fs & ::= \{\!|\, p_1 \mapsto (a_1, C_1), \ldots, p_n \mapsto (a_n, C_n) \,|\!\} \\
\text{Specifications } s & ::= k_{\tau_1}^{\tau_2} \mid e :: s \mid \langle x{:}s_1, s_2 \rangle \mid [s \mid x \in e] \mid P(e) \mid s?
\end{array}
$$

**Figure 2.2:** Forest Syntax

data. Similarly, every in-memory representation has a corresponding specification (by definition, since the former is generated from the latter), but multiple specifications map to the same in-memory representation (due to dependent types).

The authors of Forest resolve the questions raised by this disparity by ensuring that their semantics satisfies *round-tripping* laws. These are borrowed from literature on bidirectional lenses [12] and were first used to resolve the View-Update problem in databases, which concerns a similar disparity. In Forest, we can think of loading and storing functions as providing a *lens* that the data can pass through. On one side of the lens, we have the on-disk representation; on the other, the in-memory representation. The lens itself is encompassed by the specification. By passing through the lens, the data changes from one format to the other. Two round-tripping laws are used in Forest. Informally, the first states that if we load on-disk data and then immediately store it back, the file system should not change, nor should the manifest have errors. The second states that if we store an in-memory representation, whose manifest did not have errors, and then load it immediately after, we should get back what we stored.

In order to show that Forest actually obeys these laws, we must introduce some formalism. This formalism will additionally be helpful in future chapters, which extend and contrast the semantics of Forest in various ways.

Figure 2.2 contains the syntax of Forest. A path is modeled as a sequence of strings. There are three possible contents $C$ of a file system node: (1) a file with the string it contains ($\mathsf{File}\ u$); (2) a symbolic link with the path it points to ($\mathsf{Link}\ p$); or (3) a directory with the names of the set of children it contains ($\mathsf{Dir}\ \ell$). File systems are maps from paths

to attribute and content pairs. Attributes $a$ represent the metadata of a node and should be thought of as roughly the results of a POSIX `stat` command.

We only consider well-formed file systems, which model trees where every inner node is a directory:

**Definition 2.3.1** (Well-Formedness)**.** A file system $fs$ is *well-formed* if and only if:

1. $fs(/) = \mathsf{Dir}\ \_$ (where `/` is the root node), and
2. $p/u \in fs \iff fs(p) = \mathsf{Dir}\ \ell \ \wedge\ u \in \ell$

The specification surface language used in the examples above translates to the core specification language shown in Figure 2.2. Each specification $s$ describes properties about a filestore with respect to the current path in the current file system (both given by the context). The $k^{\tau_2}_{\tau_1}$ specification captures the base types of the language with $\tau_1$ and $\tau_2$ being the representation and metadata types respectively. We consider constants $File$, $Link$, and $Dir$ for files, links, and directories as representing their constant types. For example, $File = k^{\texttt{unit}}_{\texttt{string}}$ and a file at the current path conforms to that specification. Paths $e :: s$ describe filestores conforming to $s$ at the current path extended by $e$. Options $s?$ capture optional filestores: Either the current path has a filestore conforming to $s$ or it is unmapped in the file system. Dependent pairs $\langle x{:}s_1, s_2\rangle$ describe filestores that conform to both $s_1$ and $s_2$, though the subexpressions of $s_2$ can access the filestore described by $s_1$. The `directory` construct in the surface language is translated into a nested pair. Predicates $P(e)$ describe filestores for which property $e$ holds. These are usually used with dependent pairs to construct dependent types since this is the only way to refer to the data of the filestore. Finally, comprehensions $[s \mid x \in e]$ describe filestores that conform to $s$ for each value in set $e$ bound to $x$. These bound variables can change the specification $s$ by influencing its subexpressions.

Figure 2.3 defines the $\mathcal{R}[\![\cdot]\!]$ and $\mathcal{M}[\![\cdot]\!]$ semantic functions, which take a specification as input and return the corresponding representation and metadata types respectively.

| $s$ | $\mathcal{R}[\![s]\!]$ | $\mathcal{M}[\![s]\!]$ |
|---|---|---|
| $k_{\tau_1}^{\tau_2}$ | $\tau_1$ | $\tau_2$ `md` |
| $e :: s$ | $\mathcal{R}[\![s]\!]$ | $\mathcal{M}[\![s]\!]$ |
| $\langle x{:}s_1, s_2 \rangle$ | $\mathcal{R}[\![s_1]\!] * \mathcal{R}[\![s_2]\!]$ | $(\mathcal{M}[\![s_1]\!] * \mathcal{M}[\![s_2]\!])$ `md` |
| $[s \mid x \in e]$ | $\mathcal{R}[\![s]\!]$ `list` | $\mathcal{M}[\![s]\!]$ `list md` |
| $P(e)$ | `unit` | `unit md` |
| $s?$ | $\mathcal{R}[\![s]\!]$ `option` | $(\mathcal{M}[\![s]\!]$ `option`$)$ `md` |

**Figure 2.3:** Forest representation and metadata type semantic functions

For most specifications, the types are derived from their sub-specifications: Paths are ignored, pairs are pairs, comprehensions are lists, and options are options. For constants $k_{\tau_1}^{\tau_2}$ their types are included in the syntax with $\tau_1$ and $\tau_2$ respectively. Finally, predicates are not reflected in the representation. Instead, conformance to a predicate would be reflected by the lack of an error in the metadata. In the Forest calculus, the only metadata that we track is a boolean signaling errors per specification level so $\alpha$ `md` $=$ `bool` $* \alpha$.

Figure 2.4 defines the semantics of the `load` function. If loading specification $s$ in environment $E$ at a path $p$ in file system $fs$ returns the representation and metadata pair $(r, md)$, then the judgment $E \vdash$ `load` $(fs, p, s) \triangleright (r, md)$ holds. Constants $k_{\tau_1}^{\tau_2}$ have their own associated load functions, $load_k(E, fs, p)$. Path specifications $e :: s$ navigate to the path that $e$ evaluates to in $E$, before loading their sub-specification $s$. Dependent pairs $\langle x{:}s_1, s_2 \rangle$ load their first component, adding the results to the environment as $x$ and $x_{md}$, before loading the second component. The error-signaling boolean $b$, which is `true` when there are no errors, is then just the conjunction of the booleans of the sub-operations. Comprehensions $[s \mid x \in e]$ evaluate their expression $e$ into a list and load their sub-specification $s$ once per element of that list, adding the element to the environment as $x$. Predicates $P(e)$ store their value in the metadata, while options $s?$ either evaluate their sub-specification or return an empty option depending on if the path is mapped.

$$\frac{}{E \vdash \texttt{load}\ (fs, p, k^{\tau_2}_{\tau_1}) \rhd load_k(E, fs, p)}$$

$$\frac{E \vdash \texttt{load}\ (fs, [\![p/e]\!]^E_{\texttt{filepath}}, s) \rhd (r, md)}{E \vdash \texttt{load}\ (fs, p, e :: s) \rhd (r, md)}$$

$$\frac{\begin{array}{c}E \vdash \texttt{load}\ (fs, p, s_1) \rhd (r_1, md_1) \\ (E, x \mapsto r_1, x_{md} \mapsto md_1) \vdash \texttt{load}\ (fs, p, s_2) \rhd (r_2, md_2) \\ b = ((\pi_1\ md_1) \wedge (\pi_1\ md_2))\end{array}}{E \vdash \texttt{load}\ (fs, p, \langle x{:}s_1, s_2\rangle) \rhd ((r_1, r_2), (b, (md_1, md_2)))}$$

$$\frac{\begin{array}{c}[\![e]\!]^E_{\alpha\ \texttt{list}} = [w_1, \ldots, w_n] \\ \forall i \in \{1, \ldots, n\}. (E, x \mapsto w_i) \vdash \texttt{load}\ (fs, p, s) \rhd (r_i, md_i) \\ b = \bigwedge_i^n \pi_1\ md_i \qquad rs = [r_1, \ldots, r_n] \qquad mds = [md_1, \ldots, md_n]\end{array}}{E \vdash \texttt{load}\ (fs, p, [s \mid x \in e]) \rhd (rs, (b, mds))}$$

$$\frac{b = [\![e]\!]^E_{\texttt{bool}}}{E \vdash \texttt{load}\ (fs, p, P(e)) \rhd (\texttt{()}, (b, \texttt{()}))}$$

$$\frac{p \in dom(fs) \qquad E \vdash \texttt{load}\ (fs, p, s) \rhd (r, md)}{E \vdash \texttt{load}\ (fs, p, s?) \rhd (\texttt{Some}(r), (\pi_1\ md, \texttt{Some}(md)))}$$

$$\frac{p \notin dom(fs)}{E \vdash \texttt{load}\ (fs, p, s?) \rhd (\texttt{None}, (\texttt{true}, \texttt{None}))}$$

**Figure 2.4:** Forest load function semantics

Figure 2.5 defines the semantics of the `store` function. If storing $(r, md)$ as $s$ in $E$ at $p$ in $fs$ would yield a new file system $fs'$ and a validator $\varphi'$, then the judgment $E \vdash \texttt{store}\ (fs, r, md, p, s) \rhd (fs', \varphi')$ holds. The validator corresponds to the manifest that we saw earlier. Here, it is represented as a function on file systems returning `true` if the representation and metadata was successfully stored. The validator additionally checks whether the representation and metadata are consistent with each other. Constants, paths, and predicates are similar to loading. For pairs, we first get two file systems by storing each component individually, then we use the right-biased file system concatenation operator $+\!\!+$ to combine them. Intuitively, this operation copies all contents from the second argument to the first, overwriting any contents that they have in common. The

$$\frac{}{E \vdash \texttt{store}\ (fs, r, md, p, k^{\tau_2}_{\tau_1}) \rhd store_k(E, fs, p, r, md)}$$

$$\frac{E \vdash \texttt{store}\ (fs, r, md, [\![p/e]\!]^E_{\texttt{filepath}}, s) \rhd (fs', \varphi')}{E \vdash \texttt{store}\ (fs, r, md, p, e :: s) \rhd (fs', \varphi')}$$

$$\frac{\begin{array}{c} md = (b, (md_1, md_2)) \qquad r = (r_1, r_2) \\ E' = (E, x \mapsto r_1, x_{md} \mapsto md_1) \\ b' = (b = (\pi_1\ md_1) \wedge (\pi_1\ md_2)) \\ E \vdash \texttt{store}\ (fs, r_1, md_1, p, s_1) \rhd (fs_1, \varphi_1) \\ E' \vdash \texttt{store}\ (fs, r_2, md_2, p, s_2) \rhd (fs_2, \varphi_2) \\ \varphi' = (\lambda fs'.b' \wedge\ \varphi_1(fs') \wedge \varphi_2(fs')) \end{array}}{E \vdash \texttt{store}\ (fs, r, md, p, \langle x{:}s_1, s_2 \rangle) \rhd (fs_1 +\!\!+ fs_2, \varphi')}$$

$$\frac{\begin{array}{c} rs = [r_1, \ldots, r_j] \qquad mds = [md_1, \ldots, md_l] \\ [\![e]\!]^E_{\alpha\ \texttt{list}} = [w_1, \ldots, w_m] \qquad n = min(j, l, m) \\ b' = (b = \bigwedge_i^n \pi_1\ md_i) \qquad \forall i \in \{1, \ldots, n\}. \\ (E, x \mapsto w_i) \vdash \texttt{store}\ (fs, r_i, md_i, p, s) \rhd (fs_i, \varphi_i) \\ \varphi' = (\lambda fs'.(j = l = m) \wedge b' \wedge (\bigwedge_i^n \varphi_i(fs'))) \\ fs' = fs_1 +\!\!+ \ldots +\!\!+ fs_n \end{array}}{E \vdash \texttt{store}\ (fs, rs, (b, mds), p, [s \mid x \in e]) \rhd (fs', \varphi')}$$

$$\frac{\varphi' = \lambda fs'.b = [\![e]\!]^E_{\texttt{bool}}}{E \vdash \texttt{store}\ (fs, \texttt{()}, (b, \texttt{()}), p, P(e)) \rhd (fs, \varphi')}$$

$$\frac{\begin{array}{c} E \vdash \texttt{store}\ (fs, r, md, p, s) \rhd (fs', \varphi') \\ \varphi_1 = (\lambda fs'.(b = \pi_1\ md) \wedge p \in dom(fs) \wedge \varphi'(fs')) \end{array}}{E \vdash \texttt{store}\ (fs, \texttt{Some}(r), (b, \texttt{Some}(md)), p, s?) \rhd (fs', \varphi_1)}$$

$$\frac{\varphi' = (\lambda fs'.md = \texttt{None} \wedge b \wedge p \notin dom(fs'))}{E \vdash \texttt{store}\ (fs, \texttt{None}, (b, md), p, s?) \rhd (fs[p \mapsto \bot], \varphi')}$$

$$\frac{\begin{array}{c} E \vdash \texttt{store}\ (fs, r, md^s_{default}, p, s) \rhd (fs', \varphi_1) \\ \varphi' = \lambda fs'.\texttt{false} \end{array}}{E \vdash \texttt{store}\ (fs, \texttt{Some}(r), (b, \texttt{None}), p, s?) \rhd (fs', \varphi')}$$

**Figure 2.5:** Forest store function semantics

validator ensures that the boolean part of the metadata matches that of its components and that the individual validators still hold. Comprehensions use the minimum length of the representation, metadata, and expression evaluation lists to perform the storing, but check equality in the validator. Options are straightforward, except for handling cases where the representation and metadata do not match. In these cases, the validator will be false, but the file system will be updated as specified by the representation.

Using the semantics, we can now formally state the round-tripping laws as follows:

**Theorem 2.3.2** (LoadStore)**.** *Let $E$ be an environment, $fs$ and $fs'$ file systems, $p$ a path, $s$ a specification, $r$ a representation, $md$ a metadata, and $\varphi'$ a validator. If*

$$E \vdash \texttt{load}\ (fs, p, s) \rhd (r, md)$$

$$E \vdash \texttt{store}\ (fs, r, md, p, s) \rhd (fs', \varphi')$$

*then $fs = fs'$ and $\varphi'(fs')$.*

**Theorem 2.3.3** (StoreLoad)**.** *Let $E$ be an environment, $fs$ and $fs'$ file systems, $p$ a path, $s$ a specification, $r$ and $r'$ representations, $md$ and $md'$ metadata, and $\varphi'$ a validator. If*

$$E \vdash \texttt{store}\ (fs, r, md, p, s) \rhd (fs', \varphi') \qquad \varphi'(fs')$$

$$E \vdash \texttt{load}\ (fs', p, s) \rhd (r', md')$$

*then $(r, md) = (r', md')$.*

The first theorem (Theorem 2.3.2) states that loading, then storing the result, will produce an unchanged file system, which satisfies the validator. The second (Theorem 2.3.3) states that if we store a valid representation and metadata pair, we will get the same pair back if we load. This wraps up the discussion of the formalism of Forest.

We have already discussed how Forest solves several issues that we identified previously, like allowing us to avoid inconsistencies by specifying the whole filestore and checking our invariants for us, but challenges remain. Firstly, though there are

clearly advantages to the full filestore view, actually loading the whole filestore can be impractical *e.g.* due to the high cost of disk reads or because it is larger than the memory available. The original Forest partially side-steps this problem by using a lazy host language (Haskell). Unfortunately, it is sometimes difficult for users to predict what their code will load. Secondly, concurrency is pervasive in file systems and Forest offers few advantages in dealing with the issues that arise from it.

The next chapter tackles the former problem. We describe the first of the domain-specific languages designed for this dissertation, Incremental Forest. Incremental Forest allows users to manipulate and process large ad hoc filestores with precise control over the costs incurred.

Chapter 3

# Incremental Forest:

# A DSL for Efficiently Managing Filestores

*This chapter is based on joint work with Richard Zhang, Erin Menzies, Kathleen Fisher, and Nate Foster published in OOPSLA '16 [4].*

## Brief Summary

File systems are often used to store persistent application data, but manipulating file systems using standard APIs can be difficult for programmers. Forest is a domain-specific language that bridges the gap between the on-disk and in-memory representations of file system data. Given a high-level specification of the structure, contents, and properties of a collection of directories, files, and symbolic links, the Forest compiler generates tools for loading, storing, and validating that data. Unfortunately, the initial implementation of Forest offered few mechanisms for controlling cost—*e.g.*, the run-time system could load gigabytes of data, even if only a few bytes were needed. This chapter introduces Incremental Forest (iForest), an extension to Forest with an explicit *delay* construct that programmers can use to precisely control costs. We describe the design of iForest using a series of running examples, present a formal semantics in a core calculus, and define a simple cost model that accurately characterizes the resources needed to use a given

specification. We propose *skins*, which allow programmers to modify the delay structure of a specification in a compositional way, and develop a static type system for ensuring compatibility between specifications and skins. We prove soundness and completeness of the type system as well as a variety of algebraic properties of skins. We describe an OCaml implementation and evaluate its performance on applications developed in collaboration with watershed hydrologists.

## 3.1 Introduction

Previous work on Forest [7] proposed a collection of type-based abstractions for describing the structure, contents, and properties of file system data. With Forest, the programmer writes a high-level specification that describes the expected organization of a collection of directories, files, and symbolic links—a filestore—and the compiler automatically generates a datatype to represent the data in memory, accompanying *load* and *store* functions that map between on-disk and in-memory representations, and a suite of generic validation, visualization, and summarization tools. Hence, Forest allows applications to be written against high-level datatypes rather than low-level APIs, and it provides mechanisms for automatically checking assumptions about filestores.

Unfortunately, while Forest offers powerful abstractions for describing and transforming filestores, it lacks mechanisms for controlling the costs associated with using a specification, such as the amount of data read from or written to the file system, the number of file descriptors opened, and so on. A direct implementation of the language would suffer from serious performance problems. For example, it is straightforward to write a recursive *universal* specification in Forest that matches all of the files, directories, and symbolic links reachable from the root, but loading the entire file system into memory would not be feasible! Some of these issues can be side-stepped in a lazy language (the initial version of Forest was built in Haskell), but reasoning about cost remains a

challenge.

This chapter presents an extension to Forest that retains the features of the original language while offering programmers precise control over costs. Compared to the original version of the language, the main new feature provided in iForest is a *delay* construct that allows programmers to specify that certain pieces of a filestore should not be loaded or stored, unless explicitly requested by the programmer. At a technical level, a delayed specification differs from the un-delayed version in several important ways: First, rather than returning the actual value stored on the file system, the load function for a delayed specification returns a *cursor* that can be subsequently loaded (or stored) using a simple monadic interface. The types for the in-memory representation and the store function are similarly modified to reflect the fact that the value returned by the load function is a cursor and not an ordinary value. Second, a delayed specification has constant cost—*e.g.*, the load function returns immediately, without reading any data from the file system.

In general, there are many ways to add delays to a given iForest specification. With no delays, the application can manipulate values stored on the file system directly, as in Forest, but costs are coarse-grained. Alternatively, if one adds a delay at every level of the specification, then the application becomes more complicated because the type for the in-memory representation contains cursors at every level of structure. However, the cost of using the load and store functions becomes pay-as-you-go. In between these two extremes, one can add delays at different levels of granularity, making tradeoffs between the simplicity of the in-memory representation and the degree of control over costs.

To allow programmers to use the same *base* specification with different delays, we develop an expressive *skin* language that can be used to adjust the delay structure of a specification in a compositional way. We provide a parametric cost model, which can be instantiated to reason about several different types of costs. We show that costs

monotonically decrease as delays are added to a specification. We also develop a static type system that ensures compatibility between a given skin and specification.

To evaluate our design for iForest, we define a formal semantics for the language, and we prove a number of properties including round-tripping laws and natural algebraic properties of skins. We develop an iForest specification for the Soil and Water Assessment Tool (SWAT) [39], a modeling framework used by watershed hydrologists to quantify the impacts of various changes to the features of a watershed. SWAT stores persistent information in a filestore with tens of megabytes of structured files. We develop two real-world applications using iForest: one to calibrate a SWAT model against an external data set and another to predict the effects of land use changes such as changing the type of fertilizer used on farms. We conduct experiments showing that iForest leads to significant performance improvements over a naïve implementation that loads the entire filestore into memory.

Overall, the contributions of this chapter are as follows:

- We make the case for developing domain-specific tools for filestores that offer precise control over cost.
- We present iForest, a system that realizes these goals as an embedded domain-specific language in OCaml.
- We introduce skins, establish their formal properties, and show their utility on a variety of examples.
- We describe a prototype implementation of iForest and evaluate its performance on several SWAT applications.

The rest of this chapter is structured as follows. Section 3.2 motivates iForest's design and presents an overview of its main features. Section 3.3 presents the design of iForest and defines the syntax and semantics of the language. Section 3.4 introduces a

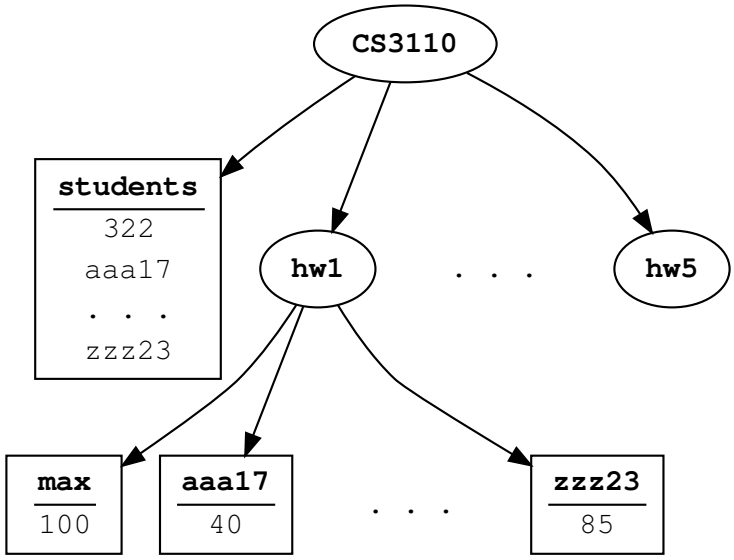



**Figure 3.1:** Example: filestore fragment used to store course data

dynamic cost model. Section 3.5 presents the skin language. Section 3.6 describes our experiences building various applications in iForest as well as quantitative experiments on SWAT data. We conclude in Section 3.7. Appendix A formalizes the main features of iForest in a core calculus and presents theorems and proofs.

## 3.2 Overview

This section motivates the design of iForest by using the running example introduced in Chapter 2. Recall the filestore fragment shown in Figure 3.1. The top-level directory (**`CS3110`**) contains a file (**`students`**) and a set of sub-directories, one for each homework assignment (**`hw1`**–**`hw5`**). The **`students`** file gives the total number of enrolled students, followed by a list of their NetIDs (Cornell's unique identifier for its students and employees). Each homework directory has a file for each student that contains their grade on the assignment (*e.g.*, **`aaa17`**), as well as a special file (**`max`**) with the maximum possible

score. There is a dependency between **students** and the homework directories—the former should contain the names of most of the files in the latter.

**Forest Implementation.** Forest [7] is a domain-specific language that provides a collection of type-based abstractions designed to support programming with filestores. With Forest, the programmer writes a specification of the expected structure, contents, and properties of a filestore. The compiler automatically generates a type for representing the data in memory, load and store functions, and a suite of other generic tools. Recapping our presentation in Section 2.3, we could use the following Forest specification to specify the structure of our example filestore:

```
hw (studentList : studentsFile_rep) = directory {
  max is "max" :: (pads pint);
  students is [student :: (pads student(max))
               | student <- matches RE "[a-z]+[0-9]+"]
              where $check_same_students this_md studentList.pstudents$}

cs3110 = directory {
  studentList is "students" :: (pads studentsFile);
  hws is [name :: hw(studentList) | name <- matches RE "hw[0-9]+"]}
```

To a first approximation, specification `cs3110` can be thought of as a type that describes the expected structure and contents of a file system at a given path. The directory construct (`directory{...}`) specifies that the file system node at the initial path should be a directory whose contents are modeled by the nested specifications. The in-memory representation for a directory is a record—in this case, with fields `studentList` and `hws` that are associated with the representations of the nested specifications. The path construct (`"students" :: (pads studentsFile)`) navigates to the specified path (**students**), while the PADS primitive (`pads`) specifies that the node at that path is represented by a PADS specification. The representation for a path is the representation for its nested specification. A comprehension specifies a collection of values—in this case, the homework directories at paths given by the regular expression `hw[0-9]+`.

The representation for a comprehension is a list. Note that the `hw` specification is parameterized by, and thus dependent on, the **`students`** file. The notation `$...$` at the end of the `hw` specification denotes that the enclosed code comes from the host language. In this case, we are ensuring that the homework directory correctly contains exactly the students listed in the **`students`** file.

Given this specification, the Forest compiler automatically generates a collection of artifacts including:

- Two types `hw_rep` and `cs3110_rep` for the in-memory representations:

```
type hw_rep = { max: pint_rep ; students: student_rep list }
type cs3110_rep = { studentList: studentsFile_rep ; hws: hw_rep list }
```

- Two types `hw_md` and `cs3110_md` for associated metadata:

```
type 'a md =
  { num_errors:int;
    error_msg:string list;
    info:file_info option;
    load_time:Time.Span.t;
    data:'a }

type hw_md =
  { max_md: pint_md md ;
    students_md: ((student_md md) list) md }
and cs3110_md =
  { studentList_md: studentsFile_md md ;
    hws_md: (hw_md list) md }
```

  where `'a` denotes a polymorphic type variable.

- Functions `cs3110_load` and `cs3110_store` that map between the on-disk and in-memory representations:

```
val cs3110_load: filepath -> (cs3110_rep * cs3110_md)
val cs3110_store: filepath -> (cs3110_rep * cs3110_md) -> unit
```

  and the same for the `hw` specification.

These functions automatically check for errors and internal inconsistencies in the data, returning useful information to the programmer even when the underlying filestore is malformed. For example, the following program implements a simple application that loads the filestore and adds a single student with NetID `student` to it:

```
let (rep,md) = cs3110_load path in
Forest.exit_on_error md;
if List.mem rep.studentList.pstudents student ~equal:String.equal
then failwithf "add_student: Student %s already exists." student ()
else
  add_student_to_filestore student (rep,md)
  |> cs3110_store
```

This implementation provides a way to gracefully handle errors. For example, if **`hw6`** were a file rather than a directory, creating an internal inconsistency in the data, we would get the following output,

```
Failure "Path <path>/hw6 is not a directory"
```

rather than a confusing run-time exception.

**Forest Limitations.** Forest's abstractions go a long way toward streamlining the task of developing applications that use file systems for storing persistent data. However, the language suffers a key limitation that makes it difficult to use in practice and leads to poor performance: it lacks mechanisms for controlling cost. If the filestore contains many large files, then naïvely loading the contents of those files into memory might exceed the resources available on the machine. A better alternative would be to allow the programmer to choose which files should be loaded eagerly and which ones should be loaded on-demand, but Forest does not provide a way to make such tradeoffs.

**Incremental Forest.** iForest is an extension to Forest that offers new mechanisms for controlling costs associated with using a given specification. The main innovation in iForest is a new *delay* construct that can be used to indicate that a certain sub-specification

should be loaded lazily rather than eagerly. For example, the following specification delays the loading of every homework specification in the **cs3110** directory by wrapping `hw(studentList)` with angle brackets, iForest's notation for the delay construct (shaded here in gray):

```
cs3110a = directory {
  studentList is "students" :: (pads studentsFile);
  hws is [name :: <hw(studentList)> | name <- matches RE "hw[0-9]+"]}
```

From this specification, the iForest compiler generates a representation type where `hws` contains *cursors* that must be explicitly *forced* to obtain the file system data.

```
type cs3110a_rep =
  { studentList: studentsFile_rep;
    hws: ((hw_rep, hw_md) cursor) list }
```

This `cs3110a_rep` type gives the programmer the flexibility to dynamically load only the files that are needed for the application. For example, to implement the same functionality as before, we could use the following program:

```
let%bind cur = cs3110a_new path in
let%bind (rep,md) = load cur in
Forest.exit_on_error md;
let%bind (rep,md) =
  if List.mem rep.studentList.pstudents student ~equal:String.equal
  then failwithf "add_student: Student %s already exists." student ()
  else add_student_to_filestore student (rep,md)
in
store cur (rep,md)
```

This program is very similar to the previous version, but has a few key differences. First, rather than having to invoke a specific load function, we use a polymorphic `load` and `store` function that takes a cursor as an argument. We create a new cursor using the function `cs3110a_new`. Second, we use a monad to keep track of the state of cursors as they are used to incrementally navigate within the file system and load data. The operator `let%bind` is a shorthand for sequencing computation with the standard monadic bind operation.

However, the most interesting part happens in `add_student_to_filestore`'s sub-routine, which maps through the homework directories, adding the given student to each one. In Forest, our sub-routine looks as follows:

```
let map_hws ~f (rep,md) =
  let (hws,hws_md) =
    let (hws,data) =
      List.zip_exn rep.hws md.data.hws_md.data
      |> List.map ~f
      |> List.unzip
    in
    (hws, {md.data.hws_md with data})
  in
  set_hws_in_cs3110 (hws,hws_md) (rep,md)
```

Here, we map over a `hws` comprehension, applying a function `f`. We first make a new `hws` representation and metadata pair by zipping the old version together, mapping over the list and subsequently unzipping it. We do it this way to make sure that we keep track of the representation and its associated metadata. Finally, we use the auxiliary function `set_hws_in_cs3110`, which properly updates the `hws` portion of the full filestore. This is simply due to the difficulty of remembering the exact record path that we want to update, as can be intuited from the long record paths in the rest of the function.

In Incremental Forest, we might instead write something like this:

```
let map_hws_inc ~f (rep,md) =
  let apply cur acc =
    acc >>= fun () -> load cur
    >>= f >>| store cur
  in
  let%map () = List.fold_right ~init:(return ()) ~f:apply rep.hws in
  (rep,md)
```

Here, our function can safely ignore the `md.data.hws_md.data` used in the previous version, because the cursors in `rep.hws` carry all of the information necessary to get their associated metadata. Thus, we fold over the cursors, and for each cursor `cur` we load it, apply function `f` and then store it back. Here, we show another way of invoking the standard monadic bind operation (`>>=`) to sequence computation and two ways of

invoking the monadic `map`. Note that we start from the accumulator each time, even though it is an empty unit. This allows continuously threading the state of our entire computation through our monad, which is important for our cost calculation. Finally, we return the representation and metadata unchanged to more closely match the signature of the previous version.

The advantage of this method is that, rather than loading every homework from the file system at once, we can load them incrementally, in a streaming fashion. This means that at any given time, the system only needs to represent the contents of a single homework in memory, and the garbage collector could reclaim the memory for previously-loaded homeworks. We could even avoid loading certain homeworks entirely—*e.g.*, those satisfying some predicate—by filtering `rep.hws` before folding over it, which would have significant performance benefits.

**Skins.** In general, there can be many different ways of adding delays to an iForest specification, depending on application needs. Some applications may wish to process file system data in larger chunks, while others may need fine-grained control over costs. Requiring programmers to write a new specification for every combination of delays that might arise would be tedious and create a software maintenance nightmare. Instead, iForest offers a *skin* language that programmers can use to modularly adjust the delay structure of an underlying *base* specification. The skin language is based on a select-and-transform paradigm in which the programmer first navigates the type structure of a given iForest specification and then manipulates the delays at that node. The primitive $\langle\,\rangle$ adds a delay while $\rangle\,\langle$ removes a delay. Because the skin language supports recursion and a rich collection of type patterns, it is relatively easy to succinctly describe many common transformations. For example, the skin

```
delayAll = ⟨⟩; map(delayAll)
```

delays every node while the skin

```
delayFiles = ⟨ ⟩|file + map(delayFiles)
```

only adds delays at `file` nodes. The `map` operator applies its sub-skin recursively, while the union operator (+) applies its left sub-skin if possible and otherwise applies its right sub-skin. The restriction operator (|) applies its sub-skin if a predicate (`file`) is satisfied. We have found skins invaluable in developing iForest applications.

## 3.3 Incremental Forest

Incremental Forest (or iForest) extends Forest with a new delay operator $\langle s \rangle$ that prevents loading (and storing) of $s$ unless explicitly forced by the programmer. This feature gives programmers precise control over costs without sacrificing the ability to write declarative filestore specifications.

At first glance, the delay operator appears quite simple. We extend the syntax of the language with delays, written $\langle s \rangle$, and let the representation and metadata types for $\langle s \rangle$ be `unit` and `unit md` respectively to reflect the fact that no processing occurs when a delayed specification is used. However, while the core elements of this design are basically right, there are a few subtle design issues that need to be addressed to make delays usable for programmers.

**Cursors.** The first issue with the design just described is that it requires programmers to manually invoke `load` functions for each delayed sub-specification forced by their application. Doing this correctly is tedious for programmers, since they will have to remember the names of the correct `load` functions to invoke. To illustrate, recall our running example and suppose that we decide to add a delay to the comprehension:

```
cs3110b = directory {
  studentList is "students" :: (pads studentsFile);
  hws is <[name :: hw(studentList) | name <- matches RE "hw[0-9]+"]>}
```

If we invoke `cs3110b_load`, we get a representation:

```
{ studentList = "10\njd753\n..."; hws = () }
```

Then, to obtain a list of homeworks, we have to invoke the `load` function for the comprehension. But now there is a problem: the comprehension does not have a name so the iForest compiler does not generate a top-level function for it! We could modify the compiler to generate `load` and `store` functions for each delayed specification, but the programmer would still need to remember the name of the function to invoke as well as the file system path to supply to that function.

To address this problem, we introduce *cursors*, which encapsulate the `load` and `store` functions associated with iForest specifications. The iForest run-time system defines a parameterized type for cursors (parameterized on the representation and metadata types) and *polymorphic* `load` and `store` functions.

```
type ('r,'m) cursor
val load : ('r, 'm) cursor -> 'r * 'm
val store : ('r, 'm) cursor -> 'r * 'm -> unit
```

Unlike the specification-specific `load` and `store` functions generated by the Forest compiler, these functions can be used with cursors of any type. To construct cursors, the iForest compiler generates a `new` function for each top-level specification `s`:

```
val s_new : filepath -> (s_rep, s_md) cursor
```

Returning to our example, if we invoke the load function:

```
load (cs3110b_new path)
```

we now get a representation

```
{ studentList = "10\njd753\n..."; hws = cur }
```

where `cur` represents the cursor for the delayed comprehension, which can be loaded:

```
load cur
```

to yield the representation:

```
[{max = 100; students = [Graded(78); Ungraded; ...]};...]
```

Cursors encapsulate the run-time details related to incremental navigation of a filestore, which greatly simplifies writing applications using iForest. To minimize overhead and support streaming computations using iForest, cursors do not cache results. We plan to investigate alternative approaches (*e.g.*, call-by-need semantics) in the future.

**Monadic Interface** Another issue with the simple design for iForest described above is that it forces results to be computed incrementally, which complicates applications. For representations and metadata, this is unavoidable—being able to operate incrementally is precisely why we designed iForest!—but it would be convenient if the run-time would aggregate other kinds of data automatically. In particular, we would like to be able to choose when data will be produced incrementally and when it will be aggregated.

To that end, we borrow a standard approach to encapsulating effectful computation from functional languages. Specifically, we define a monadic interface for iForest's `load` and `store` functions. As an example, suppose we extend the `load` function to additionally return the number of file system nodes accessed during loading, giving it the type:

```
val load : ('r,'m) cursor -> 'r * 'm * int
```

Now suppose we invoke the `load` function using the top-level and delayed cursors in sequence:

```
let cur = cs3110b_new path in
let rep1,md1,n1 = load cur in
let rep2,md2,n2 = load rep1.hws in
...
```

Note that we have to track `n1` and `n2` explicitly. We would have to compute their sum to obtain the desired result—an error-prone program structure, especially in larger applications. Instead, we can endow iForest with a monadic interface:

```
val s_new : filepath -> ((s_rep, s_md) cursor) CursorM.t
val load : ('r,'m) cursor -> ('r * 'm) CursorM.t
val store : ('r,'m) cursor -> ('r * 'm) -> unit CursorM.t
```

The module `CursorM` is a standard state monad that encapsulates the costs (represented as integers) associated with using iForest cursors:

```
module CursorM = struct
  type 'a t = int -> ('a * int)
  let return (x:'a) : 'a t = fun n -> (x,n)
  let bind (m:'a t) (f:'a -> 'a t) : 'a t =
    fun n -> let (x,n') = m n in f x n'
  let run (m:'a t) : 'a * int = m 0
end
```

With this interface (and a standard OCaml syntax extension for monads), we can re-write our example as follows:

```
let%bind cur = cs3110b_new path in
let%bind rep1,md1 = load cur in
let%bind rep2,md2 = load rep1.hws in
...
```

Now costs are encapsulated within the monad, and the aggregate value will be returned when we `run` the computation.

We can also use `CursorM` to encapsulate other kinds of state. For example, when using the `store` function incrementally, it is convenient to automatically aggregate the operations that will ultimately be executed on the file system rather than asking the programmer to keep track of them by hand.

**Dependencies.** Another issue that arises in iForest concerns dependencies. To illustrate, suppose that we revise our running example so that *both* `studentList` and `hws` are delayed:

```
cs3110c = directory {
  studentList is <"students" :: (pads studentsFile)>;
  hws is <[name :: hw(studentList) | name <- matches RE "hw[0-9]+"]>}
```

As written, this program will not compile, because `studentList` is a cursor, not a `studentsFile`, which is the type expected by the `hw` specification. To fix this error, the programmer would have to escape to our host language, explicitly `load` the cursor and

pass the result to `hw`. However, we think that this approach would be unacceptable: programmers should never have to modify a specification to accomodate delays (modulo the addition or deletion of delays). Besides being intuitive for programmers, this principle underpins our skin language (see Section 3.5). Hence, we designed iForest so that loading any cursor $c$ automatically loads any other cursors upon which $c$ depends. This design decision has a few interesting consequences:

- Any expressions occuring in a specification are written against a fully-forced specification.
- It is possible to insert useless delay annotations:

  ```
  cs3110d = directory {
    studentList is <"students" :: (pads studentsFile)>;
    hws is [name :: hw(studentList) | name <- matches RE "hw[0-9]+"]}
  ```

  Here, the delayed `studentList` is immediately forced whenever the specification is loaded. The iForest compiler accepts this specification, but emits a warning to the programmer. For simplicity, we will assume that specifications do not contain useless delays in the rest of this chapter.
- Because cursors do not currently cache data in our design, dependencies may be loaded multiple times.

Another issue related to dependencies concerns the `store` function. In general, the programmer may invoke `store` with arguments that do not satisfy the specification's dependencies. iForest currently does not check dependencies in the `store` function instead requiring the programmer to check them using the `load` function.

## 3.4 Cost Model

Incremental Forest is designed to enable programmers to make precise tradeoffs between simplicity and performance in applications that store persistent data using the file system.

| $s$ | $\mathcal{C}[\![s]\!]\ p$ |
|---|---|
| $File$ | $c_{\texttt{file}}(v)$ where $v = (\!|File|\!)\ p$ |
| $Link$ | $c_{\texttt{link}}(v)$ where $v = (\!|Link|\!)\ p$ |
| $Dir$ | $c_{\texttt{dir}}(v)$ where $v = (\!|Dir|\!)\ p$ |
| $e :: s$ | $\mathcal{C}[\![s]\!]\ (p/[\![e]\!]_e)$ |
| $\langle x{:}s_1, s_2 \rangle$ | $\mathcal{C}[\![s_1]\!]\ p \cdot \mathcal{C}[\![s_2[x \mapsto v]]\!]\ p$ where $v = (\!|s_1|\!)\ p$ |
| $[s \mid x \in e]$ | $\mathcal{C}[\![s[x \mapsto v_1]]\!]\ p \cdot \ldots \cdot \mathcal{C}[\![s[x \mapsto v_k]]\!]\ p$ where $[v_1, \ldots, v_k] = [\![e]\!]_e$ |
| $s?$ | $\begin{cases} \mathbf{0} & \text{if } \texttt{None} = (\!|s?|\!)\ p \\ \mathcal{C}[\![s]\!]\ p & \text{otherwise} \end{cases}$ |
| $P(e)$ | $\mathbf{0}$ |
| $Delay(s)$ | $\mathbf{0}$ |

**Figure 3.2:** Incremental Forest cost model

To facilitate reasoning about costs, we developed a formal model of the costs associated with using a given specification. In general, there may be a variety of costs that affect performance including the total amount of memory used, the total amount of time needed to load data into memory, the number of file system paths accessed during loading, and so on. We designed the cost model to be general—it is able to handle all of these examples, and many more.

**Formal Definition.** The cost model is parametrized on a partially ordered monoid $\mathbb{C} = \langle C, \cdot, \mathbf{0}, \sqsubseteq \rangle$, and a family of cost functions $c_\tau$ one for each primitive $\tau$ (*i.e.*, files, links, and directories). We let $p$ range over file system paths and let / denote the concatenation operator on paths. We write $v = (\!|s|\!)\ p$ if loading with $s$ at $p$ yields $v$. Similarly, we write $v = [\![e]\!]_e$ if evaluating $e$ yields $v$. We write $s[x \mapsto v]$ for the substitution of $v$ for $x$ in $s$.

Figure 3.2 presents the formal definition of the cost model. The cost $\mathcal{C}[\![s]\!]$ for each specification $s$ is defined with respect to a path $p$. The cost for a file, link, or directory

specification ($File$, $Link$, or $Dir$) is obtained by applying the corresponding primitive cost function to the representation produced by the `load` function. The cost for a path specification ($e :: s$) is simply the cost for $s$ after updating $p$ according to $e$. The cost for a pair is obtained by combining the costs for both specifications using the monoid operation ($\cdot$). However, additional care is needed to handle data dependencies: We `load` the representation of $s_1$ to $v$ and substitute it in for $x$ in $s_2$. The cost for a comprehension is obtained by combining the costs for $s$ with $v_n$ substituted for $x$, for each $n$ from $1$ to $k$. The cost for an option specification ($s?$) is $\mathbf{0}$ if the file system does not have a node at $p$ and the cost for $s$ otherwise. There is no cost for a predicate specification ($P(e)$) since $e$ is pure. Nor is there a cost for a delayed specification $Delay(s)$ since nothing is loaded once we get to the delay.

**Properties.** Given a few natural constraints, we can prove a monotonicity property for the cost model: if more delay annotations are added to a given specification, then cost monotonically decreases. Intuitively, this result holds because the iForest run-time will access fewer file system nodes. Writing $s \prec s'$ to indicate that the delays in $s$ are a subset of those in $s'$, we have the following theorem:

**Theorem 3.4.1** (Delay Monotonicity)**.** *Let $\mathbb{C} = \langle C, \cdot, \mathbf{0}, \sqsubseteq \rangle$ be a partially ordered monoid, and let $s$ and $s'$ be specifications. If $s \prec s'$ and $\forall x, y, z \in C.\, x \cdot z \sqsubseteq x \cdot y \cdot z$, then:*

$$\mathcal{C}[\![s']\!]\ p \sqsubseteq \mathcal{C}[\![s]\!]\ p$$

The reason for the requirement ($x \cdot z \sqsubseteq x \cdot y \cdot z$) is that any sub-specification may be delayed in a nested pair specification. It is important that cost does not increase just because such a delayed specification happens to lie in the middle of a group of operations. There are stronger formulations that may seem more intuitive. For example, this property can be derived if operator $\cdot$ is commutative and $x \sqsubseteq x \cdot y$. We prefer to impose this slightly less natural, but weaker requirement.

**Examples.** Incremental Forest's cost model can handle a wide variety of examples, including each of the following:

- $\mathbb{C} = (\mathbb{N}, 0, +)$, $\sqsubseteq$ is $\leq$ on the natural numbers and $c_\tau$ returns the file size for links and files and $0$ for directories. The total cost of a specification will be the sum of the sizes of all files loaded.
- $\mathbb{C} = (\mathbb{R}_+, 0, +)$, $\sqsubseteq$ is $\leq$ on the real numbers and $c_\tau$ returns the amount of time it took to load the file or link and $0$ for directories (since loading all of its components will already be taken into account). The total cost of a specification will be the sum of the load times of every file.
- $\mathbb{C} = (\mathbb{N}, 0, +)$, $\sqsubseteq$ is $\leq$ on the natural numbers and $c_{\texttt{file}}(\_) = 1$ and $c_{\texttt{link}}(\_) = c_{\texttt{dir}}(\_) = 0$. The total cost of a specification will be the number of files (not including links) loaded.
- $\mathbb{C} = (\mathbb{M}, \varnothing, \cup)$ where $\mathbb{M}$ is a multiset of files, $\sqsubseteq$ is the subset relation on multisets and $c_\tau$ returns the name of the file, link, or directory. The total cost of a specification will be the multiset containing the names of everything loaded.

All of these examples have the monotonicity property of Theorem 3.4.1. As discussed in Section 3.3, given a cost model, the cursor monad aggregates these costs automatically. Our current implementation provides a library that includes each of the four examples shown above.

## 3.5 Skins

The iForest delay construct allows programmers to control the costs associated with loading and storing file system data. However, a significant practical problem remains. Many iForest specifications are used by more than one application (or by more than one component of the same application), and these different clients can require loading

different portions of the filestore. Depending upon the details of the application, different delays may be appropriate.

Using the features we have introduced so far, iForest programmers have three options, all of which are unattractive:

1. They could only delay the parts of the specification that are not used by any application, foregoing many of the benefits of iForest.
2. They could delay every node, cluttering the application with a lot of extra code to force explicit loading.
3. They could copy the iForest specification and customize the delays for each application, duplicating code and creating a maintenance nightmare.

iForest skins provide a better, more principled way to address these problems.

Our design for skins starts from the observation that many specifications have the same underlying structure and differ only in where delays occur. A skin describes the desired pattern of delays needed for a particular application. The programmer can apply a skin to a specification to obtain a new specification that has the same structure, but different delays. Most skins make assumptions about the structure of the specifications to which they can be applied. Consequently, we define a type system for iForest specifications and skins to check their compatibility. The type system is based on fairly standard constructs for tree-structured data; the details are given in Appendix A.3.

Figure 3.3 defines the syntax of the skin language and gives types for the most important operations on skins. The delay skin ($\langle\ \rangle$) adds a delay at the top of the specification, the un-delay skin ($\rangle\ \langle$) removes a delay, and the invert skin ($\sim$) toggles a delay. The identity skin (_) does nothing. The option ($h?$), pair ($\{h_1, h_2\}$), and comprehension ($[h]$) skins modify their sub-specifications. The predicate skin ($h|\Phi$) applies $h$ only if $\Phi$ is satisfied. The union skin ($h_1 + h_2$) applies $h_1$ if possible and

$$
\begin{array}{l|l}
h ::= \langle\ \rangle & \\
\quad \mid\ \ \rangle\ \langle & \qquad s \in \mathit{Specification} \\
\quad \mid\ \sim & \qquad h \in \mathit{Skin} \\
\quad \mid\ \_ & \qquad e \in \mathit{Exp} \\
\quad \mid\ \overline{h}? & \qquad \Phi \in (\mathit{DTree} \to \mathbb{B}) \\
\quad \mid\ \{h_1, h_2\} & \\
\quad \mid\ [h] & \\
\quad \mid\ h|\Phi & \qquad [\![\cdot]\!]_h : \mathit{Skin} \to \mathit{DTree} \rightharpoonup \mathit{DTree} \\
\quad \mid\ h_1 + h_2 & \qquad \mathit{dtreeof} : \mathit{Specification} \to \mathit{DTree} \\
\quad \mid\ h_1; h_2 & \qquad \mathit{apply} : \mathit{Specification} \to \mathit{DTree} \rightharpoonup \mathit{Specification} \\
\quad \mid\ \mathit{map}(h) &
\end{array}
$$

**Figure 3.3:** Skin language syntax

otherwise applies $h_2$. The composite skin ($h_1; h_2$) applies $h_1$ and $h_2$ in sequence. The map skin ($\mathit{map}(h)$) applies $h$ to each sub-specification.

To a first approximation, a skin can be thought of as denoting a transformation that never modifies the underlying structure of the tree. We can enforce this property using the notion of a *delay tree*, which captures the paths in the specification where delays occur. Formally, a skin denotes a (partial) function on delay trees ($[\![\cdot]\!]_h$). We can extract a delay tree from a specification ($\mathit{dtreeof}$), and we can apply a delay tree to a specification to obtain a new specification ($\mathit{apply}$).

**Examples.** To illustrate the use of skins, consider the Forest specification for the running example that we have been using throughout this chapter:

```
cs3110 = directory {
  studentList is "students" :: (pads studentsFile);
  hws is [name :: hw(studentList) | name <- matches RE "hw[0-9]+"]}
```

In earlier sections, we explored how adding delays in four different configurations would affect this specification. Now we look at how users could have generated these variants by using skins instead of copying and pasting:

```
cs3110aSkin = {_,[<>]}
cs3110bSkin = hws(<>)
cs3110cSkin = {<>,<>}
cs3110dSkin = {~,_}
```

To get the four variations defined previously, we can apply these skins to the base specification as follows:

```
cs3110a = cs3110 @ cs3110aSkin
cs3110b = cs3110 @ cs3110bSkin
cs3110c = cs3110 @ cs3110cSkin
cs3110d = cs3110 @ cs3110dSkin
```

Skin `cs3110aSkin` modifies `cs3110` by first applying the identity skin (_) to the first part of the directory, then applying the delay skin (⟨ ⟩) inside the comprehension of the second part of the specification, generating:

```
cs3110a = directory {
  studentList is "students" :: (pads studentsFile);
  hws is [name :: <hw(studentList)> | name <- matches RE "hw[0-9]+"]}
```

There is no distinction between delaying the whole path construct and just its sub-specification—*i.e.* `<name :: hw(studentList)>` and `name :: <hw(studentList)>` are equivalent. The `cs3110bSkin` modifies `cs3110` by matching on `hws` and then applying the delay skin. This matching operation can be done on any named field in a directory and de-sugars into a nested skin pair. The result is:

```
cs3110b = directory {
  studentList is "students" :: (pads studentsFile);
  hws is <[name :: hw(studentList) | name <- matches RE "hw[0-9]+"]>}
```

Skin `cs3110cSkin` simply delays both parts of the directory:

```
cs3110c = directory {
  studentList is <"students" :: (pads studentsFile)>;
  hws is <[name :: hw(studentList) | name <- matches RE "hw[0-9]+"]>}
```

Skin `cs3110dSkin` flips the delay annotation on the first part of the directory (*i.e.* the `studentList`), which becomes delayed since it was not previously, and we end up with:

```
cs3110d = directory {
  studentList is <"students" :: (pads studentsFile)>;
  hws is [name :: hw(studentList) | name <- matches RE "hw[0-9]+"]}
```

Finally, consider a simpler version of our `hw`, that does not use PADS, with two delay annotations:

```
hw1 = directory {
  max is "max" :: <file>;
  students is [student :: <file>
               | student <- matches RE "[a-z]+[0-9]+"]}
```

There is a useful skin idiom that we can use to achieve this result. The idiom delays everything with a particular type, usually given as a constant. In this instance, the skin would look as follows:

```
delayFiles = <>|file + map(delayFiles)
```

This skin uses the predicate form $(h|\Phi)$, which allows users to selectively apply a skin only if the underlying specification satisfies a predicate $\Phi$. In this case, we use the built-in predicate `file`, which tests whether the type of the specification is a file. Appendix A.3 shows the collection of built-in predicates. The `delayFiles` skin also uses the union operator: If the description it is applied to is a `file`, then it simply delays it. Otherwise, it uses the $map$ construct to walk down one layer of structure (whether option, comprehension, or directory) and applies its argument there. More formally, $map(h)$ de-sugars into $[h] + h? + \{h, h\} + \_$. The `delayFiles` skin $maps$ itself, which means that it will apply itself to every sub-specification (or do nothing if it is at a leaf node). We can apply `delayFiles` to a simplified `hw` to get `hw1`.

Both the `delayAll` skin (shown in Section 3.2) and the `delayFiles` skin are good examples of skin idioms that are useful in many different applications and illustrate the expressivity of the skin language.

**Types.** We designed a standard type system for tree-structured data to check compatibility between skins and specifications (or more specifically the delay trees derived from specifications). As a side benefit, the type system gives us a convenient set of predicates for applying delays based on the types of the nodes in the delay tree. The $\Phi$ in $h|\Phi$ is often given by a type in practice.

Since delays are annotations, they do not change the type of delay trees: types are constant with respect to skin application. This property greatly simplifies the semantics of the language and reasoning about how skins will affect a specification. For example, composing skins reduces to applying them in order. Moreover, skins that have the same type can be composed in either order, without failing. Similarly, deciding which branch of a union to apply can be determined from the type of the specification.

**Properties.** We have proven a variety of useful properties about skins, types, and their relationships. Many of these properties follow by construction. Appendix A.4 gives these as theorems and lemmas, including the soundness and completeness of the type system, closure under composition for skin application, and various algebraic properties. We have proven these properties using the formal core language specified in Appendix A.3.

## 3.6 Experience

We built prototype implementations of iForest and skins, as well as a version of PADS [8], as embedded languages in OCaml. Informally, PADS is to single files what Forest is to filestores. Our implementation is approximately 4600 lines of OCaml extension points (or PPX syntax extensions) and OCaml code. We developed an iForest specification for SWAT [39] (Figure 3.6), and we built several applications in collaboration with the Cornell Soil and Water Lab, which is led by Todd Walter. We conducted experiments showing that skins can speed up load times by approximately 7x.

**SWAT Overview.** The Soil and Water Assessment Tool (SWAT) [39] is a watershed-scale, quasi-spatially distributed, hydrologic model that is used to quantify the impact of land management practices. One use of SWAT is to simulate the effects on local rivers and streams of changing the crops and fertilizers used on farms, or changing landuse,

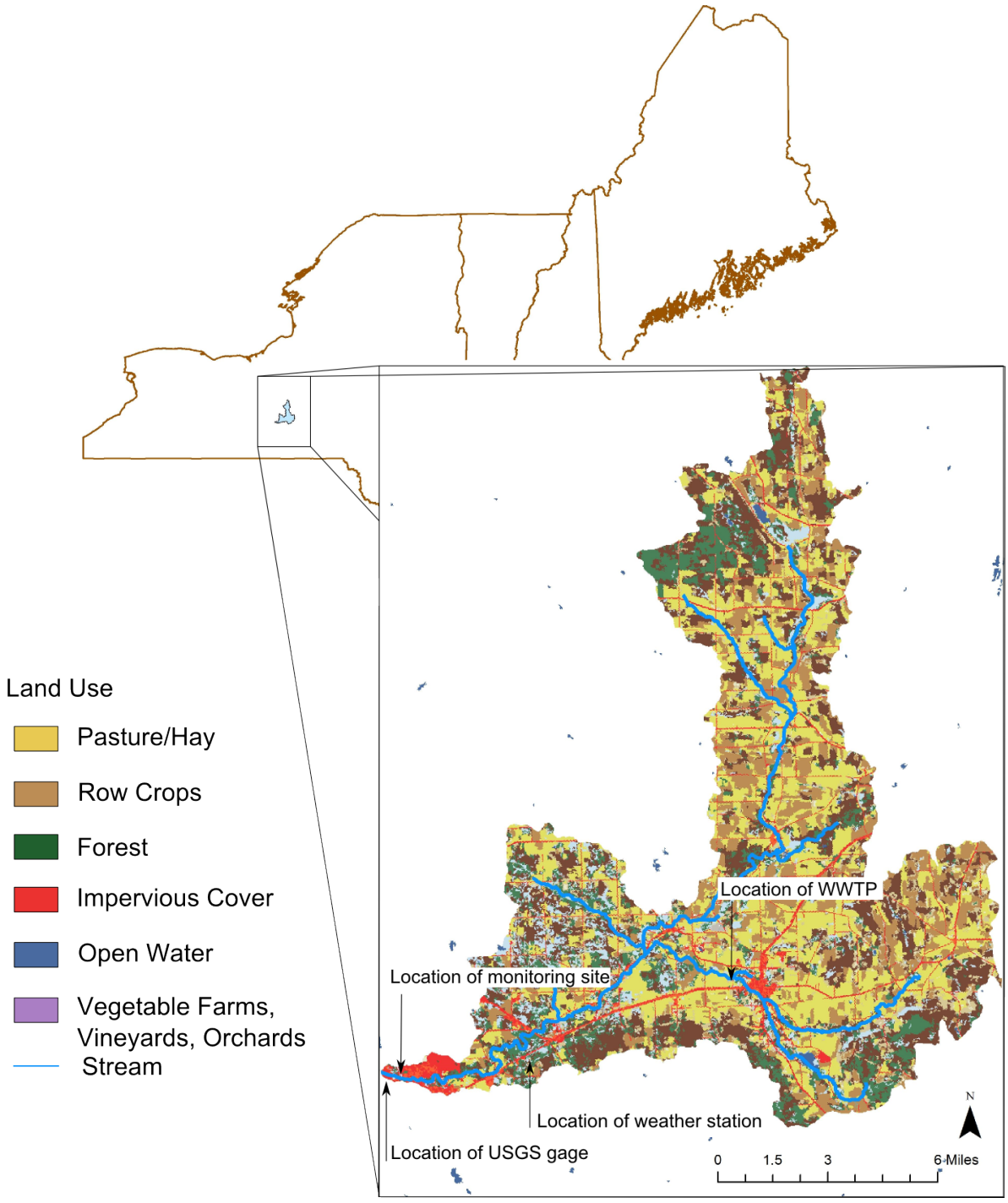



**Figure 3.4:** The Fall Creek watershed

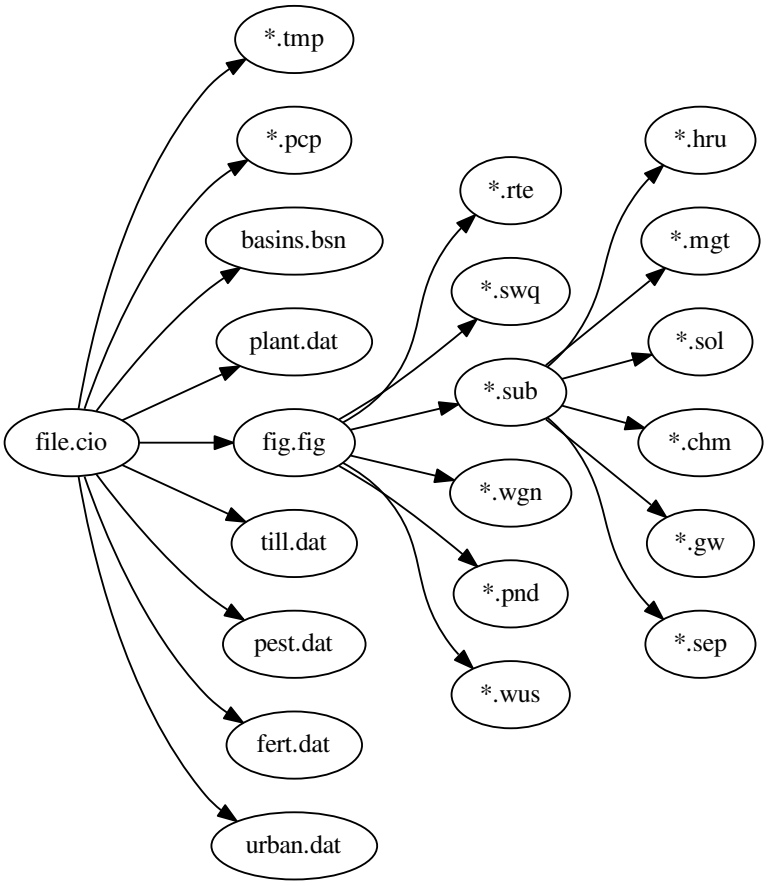



**Figure 3.5:** SWAT filestore dependencies

for example by replacing a forest with a housing development. In an instantance of SWAT, the area of interest is split into a number of non-overlapping, but contiguous subbasins, which are further broken down into Hydrologic Response Units (HRUs). HRUs are also non-overlapping, but usually *not* contiguous. However, the entirety of an HRU is identical with respect to its land use, soil types, and slope classifications even if the areas represented within are potentially far apart. Note that an HRU cannot be spread over multiple subbasins. In our examples, we used a SWAT instance from the Fall Creek watershed in Ithaca, NY, pictured in Figure 3.4.

**SWAT Filestore.** Like many similar tools, SWAT stores its persistent data using a structured filestore. A top-level directory **`TxtInOut`** contains a master index file **`file.cio`** that refers to a large number of data files (around 10,000 in our example), identified by specific names and extensions. The data files contain a variety of information about the watershed including general features, such as snowfall temperature, soil evaporation factor, and surface runoff time, as well as features specific to each sector of land in the model, such as the type of crop, fertilizer, and irrigation. Figure 3.5 depicts the various components in a SWAT filestore and the dependencies between them. Note that the names of certain nodes (**`*.hru`**) are parametrized to indicate that they are instantiated multiple times, one for every sector of land in the model. A typical SWAT filestore has thousands of files with tens of megabytes of data or more, depending on the level of detail in the model.

**Example Application: Calibration.** An important first step in any SWAT application is to calibrate the model to ensure that it accurately reflects watershed features. To do this, a scientist explores the parameter space, adjusting values within specified bounds to optimize a global objective such as Nash–Sutcliffe efficiency [33]. Concretely, calibration entails modifying input parameters stored in ASCII text files, (re)running the SWAT executable to compute derived data, and then comparing the output values, which are also stored in ASCII text files. This process is iterated many times until the optimal set of parameters, or a close approximation, is found.

**Example Application: Management.** After calibrating, many applications can be built using SWAT. One common use is quantifying the impact of various land management decisions on a watershed [13, 16, 20, 34, 42]. This involves encoding management decisions as inputs to the model and then interpreting model output. Operationally, this application is similar to calibration in that the scientist modifies input parameters stored

```
swatIn = directory {
  cio   is "file.cio"          :: cioFile;
  fig   is $cio.figFile.str$ :: figFile;
  cst   is $cstFile cio$       :: file option;
  wnd   is $slrFile cio$       :: wnd option;
  rh    is $rhFile cio$        :: rh option;
  slr   is $slrFile cio$       :: slr option;
  bsn   is $basinFile cio$     :: bsn;
  plant is $plantFile cio$     :: crop;
  till  is $tillFile cio$      :: till;
  pest  is $pestFile cio$      :: pest;
  fert  is $fertFile cio$      :: fert;
  urban is $urbanFile cio$     :: urban;
  pcps  is [ f :: pcp | f <- $pcpFiles cio$ ];
  tmps  is [ f :: tmp | f <- $tmpFiles cio$ ];
  subs  is [ f :: sub | f <- $subFiles fig$ ];
  rtes  is [ f :: rte | f <- $rteFiles fig$ ];
  swqs  is [ f :: swq | f <- $swqFiles fig$ ];
  hrus  is [ f :: hru | f <- $allHruFiles subs$ ];
  mgts  is [ f :: mgt | f <- $allMgtFiles subs$ ];
  sols  is [ f :: sol | f <- $allSolFiles subs$ ];
  chms  is [ f :: chm | f <- $allChmFiles subs$ ];
  gws   is [ f :: gw  | f <- $allGwFiles subs$ ];
  seps  is [ f :: sep | f <- $allSepFiles subs$ ];
  wgns  is [ f :: wgn | f <- $allWgnFiles subs$ ];
  pnds  is [ f :: pnd | f <- $allPndFiles subs$ ];
  wuss  is [ f :: wus | f <- $allWusFiles subs$ ] }
```

**Figure 3.6:** SWAT filestore specification. The constants in the specification that are not *File* are PADS specifications describing the contents of the individual files.

in ASCII text files, runs SWAT, and then looks at the output values in more ASCII text files.

**Forest SWAT Specification.** Forest facilitates implementing these kinds of SWAT applications. Figure 3.6 gives a Forest specification for SWAT filestores. The top-most specification is a directory that matches the top-level **`TxtInOut`** directory. The first entry is for `file.cio`, which serves as the master index for the filestore. The rest of the entries use options and comprehensions to describe the structure of the remaining files. Note that dependencies can be expressed by simply referring to values—*e.g.*, the list of PCP

files `pcps` depends on the values in the representation of `file.cio`.

**Incremental Forest SWAT Specifications.** SWAT is a large model with a host of inputs and outputs, but a given application often needs to inspect only a small set of files. The relevant files vary from application to application, however. Using skins, scientists can restrict their attention to the portion of the data in which they are interested, while sharing a single iForest description across many different applications.

**Results in Brief.** Writing an initial specification can be time consuming because of the myriad details (see the SWAT manual [38]), once we had the specification, writing applications against it was generally straightforward. We found the skin language expressive enough to describe in a few lines everything we needed. Designing a skin typically required only a few minutes. We found reasoning about skins to be mostly straightforward.

We ran experiments on data from the Fall Creek watershed in Ithaca, NY on a cluster of 24 Dell r620 servers, each with two eight-core 2.60 GHz Xeon CPU E5-2650 processors and 64GB of RAM running Ubuntu 14.04.1 LTS. We report all times in seconds unless otherwise indicated. We found that iForest yields speed-ups of approximately 7x for loads.

### 3.6.1 Microbenchmark

To get a sense of the performance improvements possible with skins, we ran a microbenchmark that quantified the time to load data from a SWAT directory using several different skins (Figure 3.7). We used the specifications in Figure 3.8 with 5 different levels of skins to load the input and output files of a 95MB SWAT directory containing 9771 files:

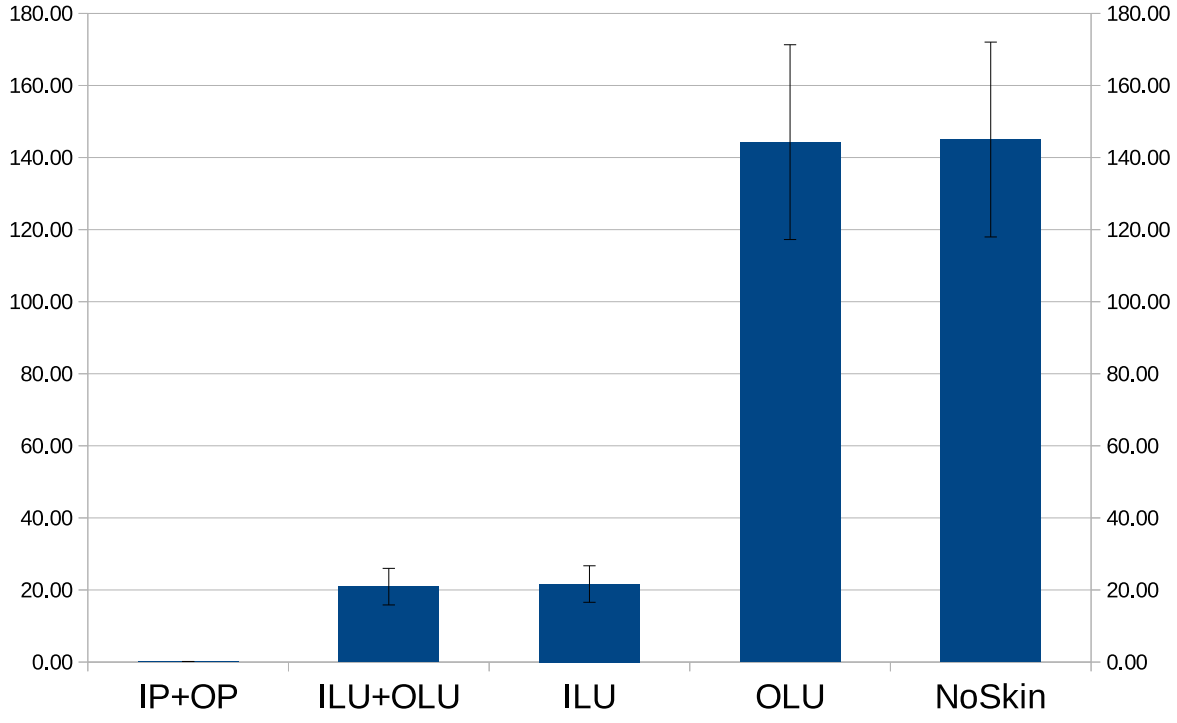
- IP+OP, which used the `swatIP` and `swatOP` specifications to load only the dependencies required for the rest of the skins. IP and OP stand for Input/Output with Predicates.
- ILU+OLU, which used the `swatILU` and `swatOLU` specifications to load exactly what is used in the land management application. ILU and OLU stand for Input/Output with LandUse.
- ILU, which used the `swatILU` and `swatOut` specifications; The `swatOut` specification is entirely undelayed and specifies the seven output files of a SWAT execution.
- OLU, which loaded the `swatIn` and `swatOLU` specifications respectively. The `swatIn` specification is entirely undelayed.
- NoSkin, which loaded the `swatIn` and `swatOut` specifications.

Figure 3.7 shows the results of the experiment, reporting the average and standard deviation of the various loading times in seconds. The ILU+OLU skinned version is roughly 7 times faster than the NoSkin unskinned version on average. The error bars show standard deviations in all charts.

### 3.6.2 Calibration

Next, we built an application that automatically calibrates a SWAT model with respect to a set of parameters. As discussed above, this is a critical first step in any SWAT application. The calibration used by our colleagues attempts to match the daily outflow of water shown in the model with the ground truth data, measured by a US Geologic Survey gauging station. Accuracy is measured using the Nash-Sutcliffe Model Efficiency (NSE) Coefficient [33],

$$E = 1 - \frac{\Sigma_{t=1}^{T}(Q_o^t - Q_m^t)^2}{\Sigma_{t=1}^{T}(Q_o^t - \bar{Q}_o)^2}$$




| Field | IP+OP | ILU+OLU | ILU | OLU | NoSkin |
|---|---|---|---|---|---|
| Avg | 0.15 | 20.92 | 21.65 | 144.28 | 145.01 |
| Stdev | 0.005 | 5.07 | 5.07 | 27.03 | 27.03 |

**Figure 3.7:** Load times for SWAT input-output directories

where $Q_o^t$ is observed discharge, $Q_m^t$ is measured discharge, $\bar{Q}_o$ is the mean of observed discharges, $t$ denotes time, and $E$ ranges from $1$ to $-\infty$. If $E = 1$, the model perfectly predicts the observations (which is extremely unlikely to arise in practice). If $E < 0$, then we would have done better by predicting the average of the observed data at every point. Generally, $E > 0.5$ is considered satisfactory [32].

Figure 3.9 lists a set of parameters that are relevant for calibrating the Fall Creek watershed, showing that the search space is extremely large. In our application, we only modified 4 parameters, chosen because they are especially sensitive: ALPHA_BF, GW_DELAY, SURLAG, and ESCO. We picked 6 points per parameter, distributed relatively evenly over the search space, and ran calibration. Specifically, we wrote all combinations

```
(* skins *)
delayAll = <>;map(delayAll)

predSkin = delayAll;fig(><);subs(><;[><]);
           cio(><)

inCalib = predSkin;bsn(><);gws(><;[><]);
          hrus(><;[><])
outCalib = delayAll;outRch(><)

inLU = predSkin;mgts(><;[><])
outLU = delayAll;outStd(><)

(* specifications *)
swatICB = swatIn @ inCalib
swatOCB = swatOut @ outCalib

swatILU = swatIn @ inLU
swatOLU = swatOut @ outLU

swatIP = swatIn @ predSkin
swatOP = swatOut @ delayAll
```

**Figure 3.8:** SWAT Skins and the resulting iForest specifications. The `swatIn` specification appears in Figure 3.6; `swatOut` is not shown.

| Parameter | Min | Max | Init | Best |
|---|---|---|---|---|
| GW_DELAY | 0.50 | 1000.00 | 31.000 | 82.410 |
| ALPHA_BF | 0.10 | 1.00 | 0.0480 | 0.152 |
| GWQMN | 0.00 | 500.00 | 0.0000 | 29.154 |
| GW_REVAP | 0.00 | 0.20 | 0.0200 | 0.192 |
| REVAPMN | 0.00 | 500.00 | 1.0000 | 443.955 |
| RCHRG_DP | 0.00 | 1.00 | 0.0500 | 0.107 |
| SFTMP | -5.00 | 5.00 | 1.000 | -0.424 |
| SMTMP | -5.00 | 5.00 | 0.500 | 3.286 |
| SMFMX | -5.00 | 5.00 | 4.500 | 1.843 |
| SMFMN | -5.00 | 5.00 | 4.500 | 3.611 |
| TIMP | 0.00 | 4.00 | 1.000 | 0.553 |
| SURLAG | 0.00 | 15.00 | 4.000 | 0.246 |
| ESCO | 0.10 | 1.00 | 0.950 | 0.583 |
| EPCO | 0.00 | 1.00 | 1.000 | 0.955 |
| NSE | $-\infty$ | 1.00 | -1.034 | 0.621 |

**Figure 3.9:** Calibration parameters

to the input files, running SWAT with each combination, and recorded the best values.

With this approach, we achieved an NSE of 0.41. It is worth noting that when we combined the best values for our four parameters with the best values for all other parameters previously found by hydrologists and reran SWAT, we got an NSE of 0.625. This value is slightly *better* than the value of 0.621 that our colleagues had previously obtained.

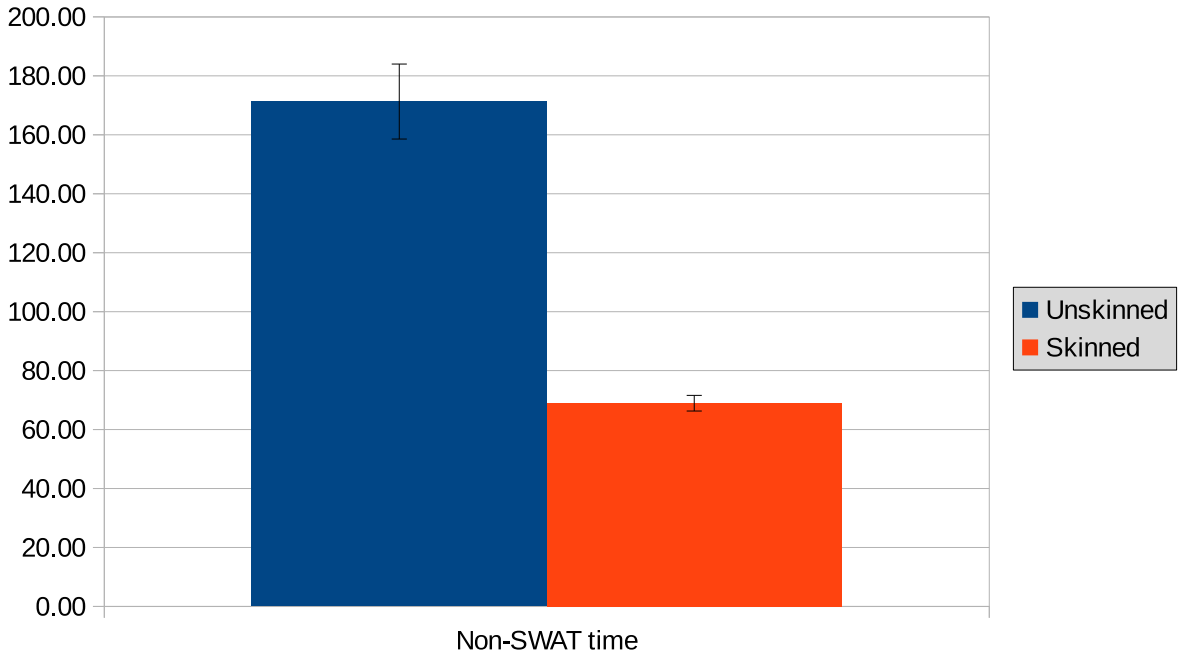
Figure 3.10 gives the running time of our calibration application. The majority of the time comes from running SWAT, which takes circa 2.5 minutes per run. This executable is a black box so we can do nothing to improve it. We see smaller improvements than the speedup measured earlier, even in Non-SWAT time, presumably because the program is no longer just loading. Even so, the skinned version is notably faster.

### 3.6.3 Land Management

Another common use of SWAT is to simulate the impact of various land management decisions, such as which crops to plant and when, which water sources to irrigate from and how much, or what fertilizers to use and when, *etc*. [13, 16, 20, 34, 42]. This is done by modifying management input files describing which decisions should be simulated in the model. There is one such file for each HRU in a SWAT directory.

To show that iForest can handle such situations, we built an application that systematically changes some management input files, runs SWAT, and then looks at selected output parameters to observe the results. Specifically, we changed the fertilizer to Fresh Dairy Manure, ran SWAT, and then looked at how the amount of Organic Nitrogen in the water varies with the amount of fertilizer. This approach is sufficiently generic that only small changes would be required to switch what parameters to change, how to change them, where to change them, and what output value to track.

Figures 3.11 and 3.12 show our results. The first figure reports timing information,



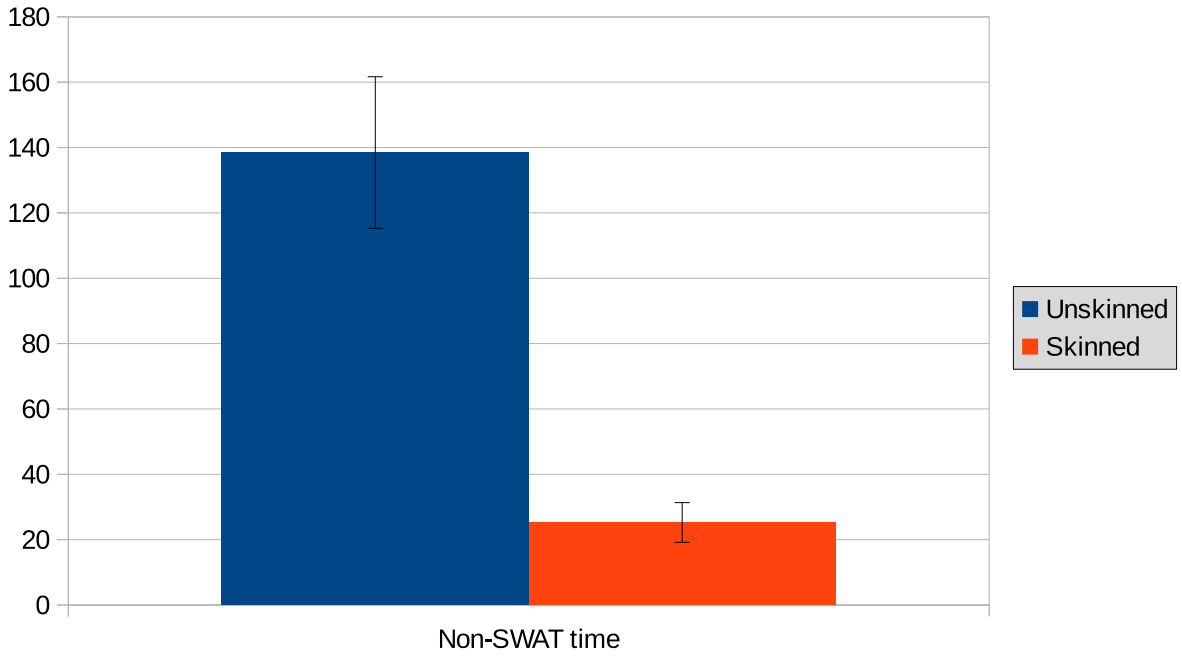

| **Unskinned** | Total Time | Non-SWAT time | SWAT time |
|---|---|---|---|
| Average | 1054.89 | 171.29 | 883.60 |
| Stdev | 16.16 | 12.72 | 8.19 |
| **Skinned** | | | |
| Average | 943.17 | 68.94 | 874.23 |
| Stdev | 6.02 | 2.65 | 5.36 |

**Figure 3.10:** Calibration experiment

showing that we obtained a 5.5x speedup in non-SWAT time. The second depicts how organic nitrogen in the stream increases as we use more Fresh Dairy Manure. The curve is not smooth because the HRUs behave differently.

## 3.7 Conclusion

This chapter presented Incremental Forest, a domain-specific language that extends Forest [7] with a new delay construct to enable processing file system data incrementally.




| **Unskinned** | Total Time | Non-SWAT time | SWAT time |
|---|---|---|---|
| Average | 1766.13 | 138.48 | 1627.64 |
| Stdev | 24.26 | 23.20 | 6.55 |
| **Skinned** | | | |
| Average | 1668.72 | 25.25 | 1643.47 |
| Stdev | 11.38 | 6.07 | 8.59 |

**Figure 3.11:** Land management experiment

We described the delay construct as well as a cursor type for encoding it. We introduced a generic cost model and showed that costs monotonically decrease as delays are added, subject to natural conditions. We described a language of skins, which allows programmers to induce different delay annotations in specifications without rewriting them. We presented a type system to ensure that skins are only applied to compatible specifications, and we proved the type system sound and complete. Finally, we described case studies based on the Soil and Water Assessment Tool (SWAT), which hydrologists use to study watersheds. Specifically, we discussed a calibration application and a

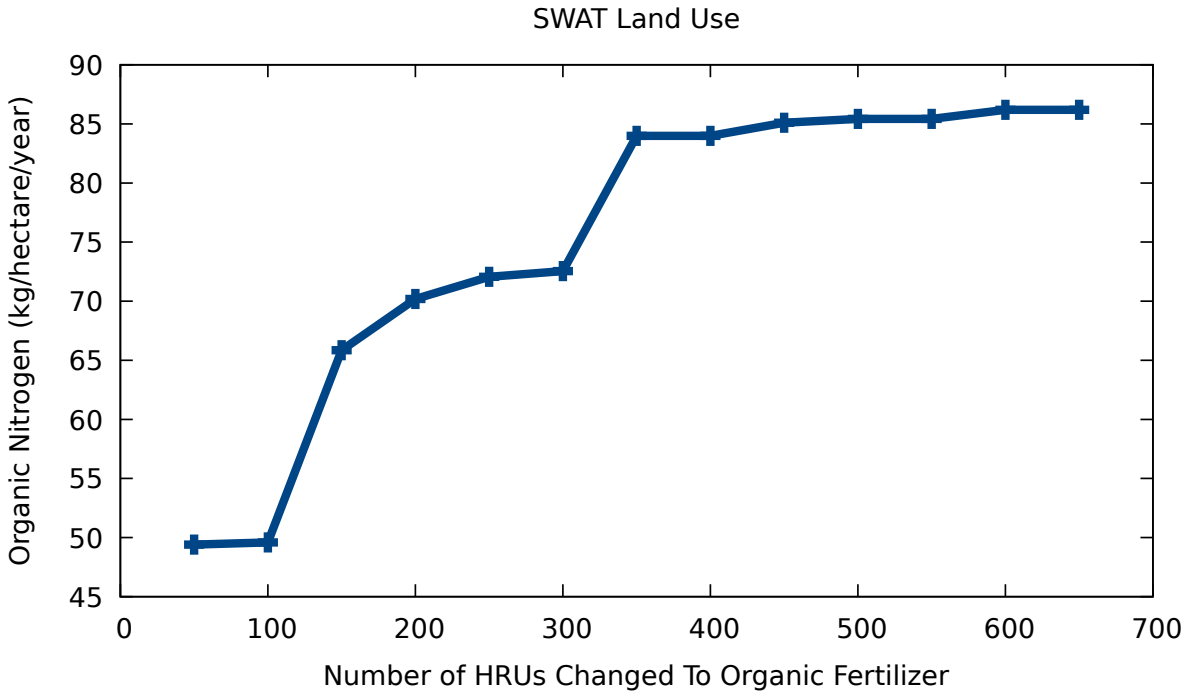



**Figure 3.12:** Increase in organic nitrogen as more HRUs use Fresh Dairy Manure fertilizer

management application that we have written using the iForest SWAT specification. We also reported performance results on a microbenchmark showing a speedup of 7x when loading with a skin versus loading naïvely.

Unfortunately, although Incremental Forest deals with the issue of large filestores, it has little support for concurrency. The next chapter will be a brief primer on concurrency and zippers in preparation for Transactional Forest (Chapter 5), a system that we designed to tackle precisely the concurrency problem.

Chapter 4

# On Concurrency and Zippers:

# A Brief Interlude

In this chapter, we explore concurrency issues that may appear in our file systems, and we discuss zippers. That sets the scene for the next two chapters.

## 4.1 Concurrency

Concurrency is ubiquitous in file systems and arises from two separate sources: (1) Multiple concurrent users of the file system (*e.g.* in a networked file system); and (2) parallel processes from single users. Unfortunately, concurrency increases the difficulty of writing correct code. As programmers, we want to think of our applications as running in isolation, making the relevant state of our system predictable during the execution of our programs. A concurrent system may instead change behind our back, breaking this invariant, and complicating reasoning about our code. This section will introduce several common concurrency-related problems and common high-level approaches for tackling them.

Concurrency errors can arise when two transactions $t_1$ and $t_2$ that access or modify the same data are run at the same time. The errors stem from unfortunate interleavings

of these operations, because transactions are not isolated from each other. The data that multiple transactions have access to is called the *shared state* of these transactions. At first approximation, there are three access/modification patterns that lead to difficulties called *conflicts* between transactions:

1. **Non-Repeatable Reads** arise when $t_1$ reads the data and $t_2$ modifies it before $t_1$ reads it again. From $t_1$'s view, it should be reading the same data in both circumstances, but instead the data has changed. This is also called a **Read-Write** conflict.
2. **Dirty Reads** arise when $t_1$ writes data and $t_2$ reads it before $t_1$ writes it again. Then, $t_2$ sees intermediate data that $t_1$ never intended to be seen and that would never be seen if the transactions were run sequentially. For example, in the case of filestores, $t_2$ can see an inconsistent state which violates our invariants. This is also called a **Write-Read** conflict.
3. **Dirty Writes** arise when there are at least two pieces of data, `A` and `B`. If $t_1$ writes `A`, then $t_2$ writes both, then $t_1$ writes `B`, we end up with $t_1$'s `B` and $t_2$'s `A`. This may be an inconsistent state, which, again, may violate our invariants. This is also called a **Write-Write** conflict.

All of these issues arise because transactions are not *isolated* from each other—each transaction can observe the effects of partial execution by others. This causes our transactions to violate the principles of *atomicity* and *consistency*. Atomicity requires that the effects of a transaction either happen all-at-once or not-at-all: the effects of the transaction are indivisible. Consistency requires that a transaction that starts in a consistent state should also finish in a consistent state. As we've seen above, neither of these properties hold due to the conflicts arising from a lack of isolation.

There are traditionally four guarantees that we desire in a concurrent system, which are expressed with the acronym ACID [21]. These are **A**tomicity, **C**onsistency, **I**solation,

and **D**urability. The first two were described above. Isolation guarantees that transactions cannot interfere with one another, while durability ensures that the effects of transactions persist through crashes. Atomicity and durability together ensure that either no effect or the full effect of a transaction persists through a crash.

The overhead of maintaining ACID increases the longer a transaction runs. Simultaneously, many parts of a program do not touch any of the shared state between transactions, and thus can be safely ignored for the purposes of maintaining ACID. Therefore, rather than considering a full program as a transaction, it behooves us to split programs into multiple pieces, some of which are transactions and some of which are not. This split allows other concurrent operations to be executed between the transactions, reducing the overhead while maintaining ACID it is needed. A transaction's full effect should be durably represented on the file system when it is *committed*. Alternatively, a transaction can also be *aborted*, in which case none of its effects should persist.

*Concurrency Control Schemes* are strategies for enforcing ACID guarantees in concurrent systems. There are two main approaches to concurrency control: pessimistic and optimistic.

Pessimistic concurrency control embodies the view that transactions are likely to conflict, so we should limit them a priori to avoid those conflicts. Usually, this is achieved by requiring that transactions lock shared resources before using them, ensuring that other transactions cannot concurrently access the shared resource. Sloppy designs or implementations of pessimistic strategies can result in *deadlocks*. Deadlocks occur when two (or more) transactions are waiting for resources that the other transaction has already locked, so neither can make progress. The use of locks limits the concurrency that a system can exploit, since multiple transactions cannot access the same data—even if they would not conflict in the end. On the other hand, it can be easier to determine whether our desired ACID guarantees hold with pessimistic concurrency control, since

transactions cannot run concurrently if they access the same resource. Two-phase locking [18] (2PL) is a common pessimistic concurrency control strategy.

Optimistic concurrency control emboides the opposite philosophical position: we presume that most transactions will not conflict, so we should not limit their execution, but instead check for conflicts before committing a transaction's results. This semantics is often achieved by logging the effects of each transaction locally, then checking for a conflict with any committed transaction by comparing their local log to a global log. If there is no conflict, then the local log is integrated into the global log, and the transaction is committed. If there was a conflict, then the transaction is aborted (and often restarted). If there are no conflicts and the overhead of logging and log-checking is low, then this approach can be very efficient. On the other hand, if there are many conflicts and transactions have to be restarted frequently, there will be much duplicate work. Commit Ordering (CO) is a common optimistic concurrency control strategy.

The key property that we will enforce between transactions in this dissertation is called *serializability*. It ensures that the result of any concurrent set of transactions will be the same as though they had run in sequential order. Regardless of whether the transactions are actually atomic or isolated, serializability ensures that, at the system level, they behave as though they were. Additionally, if each transaction maintains consistency and durability, a serializable system will do the same. Variants of serializability are the strongest guarantees in common use in database systems.

In the next chapter, we use an optimistic concurrency scheme to achieve serializability among Forest transactions, mitigating concurrency issues in filestores. We use a similar scheme to construct a transactional file system with provable guarantees in Chapter 6. Both chapters also use the zipper data structure from functional programming, which we introduce in the next section.

## 4.2 Zippers

*Zippers* are a functional representation of in-progress traversals of data structures. Zippers are derivable from the structures that they represent. This section discusses why tree zippers are a good fit for representing filestores. We also present the tree zipper instantiation that we use throughout the dissertation.

The concept of zippers was first published by Huet [23]. Huet identifies, what we would now call, a tree zipper, suggesting that it must have been reinvented on numerous occasions due to the simplicity and elegance of the idea. In essence, the zipper has two components; the current subtree of focus and the path taken from the root of the full tree to reach that subtree. The key goal is to enable tree updates without relying on destructive mutation or requiring logarithmic complexity for each update.

The zipper achieves this by supplying a set of local (constant time) operations for moving through the tree and updating the focus node. For example, we might `go_down` from the root node to its first child, making that the focus node, then `go_right` to its sibling (*i.e.* the second child of the root). We could `change` the new focus node to be a leaf, effectively deleting that subtree, then `go_up` to return to the root. Notably, if we design the two components (the focus and the path) correctly, then each of the traversal operations (*i.e.* those starting with `go`) is just a small reshuffling of the zipper. The `change` operation simply swaps out the current focus node with a new one. Finally, while any changes would not persist in the original tree, since it is a functional structure, we can quite easily use the `go_up` operation to properly propagate any changes that we have made. This means that we can recover the modified tree by moving back to the root and extracting the focus node.

Since then, the concept of zippers has been generalized to other structures, capturing their in-progress traversals in a functional data structure. In work that I find particularly

compelling, McBride showed a correspondence akin to the standard calculus derivative between zippers and the structures that they capture [31].

However, for the rest of this dissertation, we will restrict attention to tree zippers, since they are well-suited to representing file systems. In particular, the notion of a working path is neatly encompassed by the zipper. The focus node captures the fact that the commands are executed relative to this working path, and the path component of the zipper, much like the literal working path, shows from whence we came.

In Chapters 5 and 6, we use zippers represented by this OCaml type:

```
type 'a zipper =
  { ancestor: 'a zipper option;
    left: 'a list;
    focus: 'a;
    right: 'a list;
  }
```

The particular instantiation of `'a` varies by chapter, but the rest is the same. The `ancestor` uses a nesting of zippers to represent the path from the root to the focus node. If the root is the focus node, the `ancestor` will be `None`. The siblings to the left and right of the focus node are captured by the lists `left` and `right` respectively. The `left` list is stored in reverse, with its closest sibling as its first element. Finally, `focus` is the focus node.

Due to the verbosity of defining each zipper using the syntax above, we instead use a special notation:

**Definition 4.2.1** (Zipper Notation)**.** We define notation for constructing and deconstructing (*i.e.* pattern matching on) zippers. To construct a zipper we write:

$$\texttt{left} \leftharpoonup \wr \texttt{focus} \int^{z} \rightharpoonup \texttt{right} \triangleq \{\texttt{ancestor} = \texttt{Some}(z); \texttt{left}; \texttt{focus}; \texttt{right}\}$$

where any of `ancestor`, `left`, and `right` can be omitted to denote a zipper with $\texttt{ancestor} = \texttt{None}$, $\texttt{left} = []$, and $\texttt{right} = []$ respectively. For example:

$$\wr \texttt{focus} \int \triangleq \{\texttt{ancestor} = \texttt{None}; \texttt{left} = []; \texttt{focus}; \texttt{right} = []\}$$

Likewise, to destruct a zipper we write:

$$\texttt{left} \leftharpoonup (\!|\texttt{focus}|\!)^{z} \rightharpoonup \texttt{right}$$

where any part can be omitted to ignore that portion of the zipper, but any included part must exist. For example, $z = (\!|\_|\!)^{z'} \iff z.\texttt{ancestor} = \texttt{Some}(z')$.
Finally, to construct a new zipper from an existing zipper and a focus node, we define:

$$z \textit{ with } f \triangleq \{z \texttt{ with focus} = f\}$$

This covers the majority of the specialized knowledge required to understand the next two chapters. In the next chapter, we look at Transactional Forest, a domain-specific language and system that we designed for processing concurrent filestores.

Chapter 5

# Transactional Forest:

# A DSL for Managing Concurrent Filestores

*This chapter is based on joint work with Katie Mancini, Kathleen Fisher, and Nate Foster published in ASPLAS '19 [3].*

## Brief Summary

Many systems use ad hoc collections of files and directories to store persistent data. For consumers of this data, the process of properly parsing, using, and updating these *filestores* using conventional APIs is cumbersome and error-prone. Making matters worse, most filestores are too big to fit in memory, so applications must process the data incrementally while managing concurrent accesses by multiple users. This chapter presents Transactional Forest (TxForest), which builds on earlier work on Forest and iForest to provide a simpler, more powerful API for managing filestores, including a mechanism for managing concurrent accesses using serializable transactions. TxForest implements an optimistic concurrency control scheme using Huet's *zippers* to track the data associated with filestores. We formalize TxForest in a core calculus, develop a proof of serializability, and describe our OCaml prototype.

## 5.1 Introduction

In many applications, multiple users must read and write the data stored on the file system concurrently. Even in settings where there is only a single user, parallelism can often be used to improve performance. For example, many instructors in large computer science courses rely on filestores to manage student data, encoding assignments, rosters, and grades as ad hoc collections of directories, CSVs, and ASCII files respectively. During grading, instructors use various scripts to manipulate the data—computing statistics, normalizing raw scores, and uploading grades to the registrar. However, these scripts are written against low-level file system APIs and rarely handle multiple concurrent users. This can easily lead to incorrect results or even data corruption in courses that use large numbers of TAs to help with grading.

The PADS and Forest family of languages, described in earlier chapters, offers a promising approach for managing ad hoc data. In these languages, the programmer specifies the structure of an ad hoc data format using a simple, declarative specification, and the compiler generates an in-memory representation for the data, load and store functions for mapping between in-memory and on-disk representations, as well as tools for analyzing, transforming, and visualizing the data. PADS focuses on ad hoc data stored in individual files [8], while Forest handles on ad hoc data in *filestores*—*i.e.*, structured collections of files, directories, and links [7]. Unfortunately, these languages lack support for concurrency.

To address this challenge, this chapter proposes Transactional Forest (TxForest), a declarative domain-specific language for correctly processing ad hoc data in the presence of concurrency. Like its predecessors, TxForest uses a type-based abstraction to specify the structure of the data and its invariants. From a TxForest description, the compiler generates a typed representation of the data as well as a high-level programming interface

that abstracts away direct interactions with the file system and provides operations for automatically loading and storing data, while gracefully handling errors. TxForest also offers serializable transactions to help implement concurrent applications.

The central abstraction that facilitates TxForest's serializable semantics, as well as several other desired properties, is based on Huet's *zippers* [23]. Rather than representing a filestore in terms of the root node and its children, a zipper encodes the current node, the path traversed to get there, and the nodes encountered along the way. Importantly, local changes to the current node as well as common navigation operations involving adjacent nodes can be implemented in constant time. Additionally, by replacing the current node with a new value and then 'zipping' the tree back up to the root, modifications can be implemented in a purely functional way.

As others have also observed [27], zippers are a natural abstraction for filestores, for several reasons: (1) the concept of the working path is cleanly captured by the current node; (2) most operations are applied close to the current working path; (3) the zipper naturally captures incrementality by loading data as it is encountered in the zipper traversal; and (4) a traversal (along with annotations about possible modification) provides all of the information necessary to provide rich semantics, such as copy-on-write, as well as a simple optimistic concurrency control scheme that guarantees serializability.

In this chapter, we first formalize the syntax and semantics of TxForest assuming a single thread of execution and establish various correctness properties, including round-tripping laws in the style of lenses [12]. Next, we extend the semantics to handle concurrent threads and introduce a transaction manager that implements a standard optimistic concurrency control scheme. We prove that all transactions that successfully commit are serializable with respect to one another. Finally, we present a prototype implementation of TxForest as an embedded language in OCaml, illustrating the feasibility of the design.

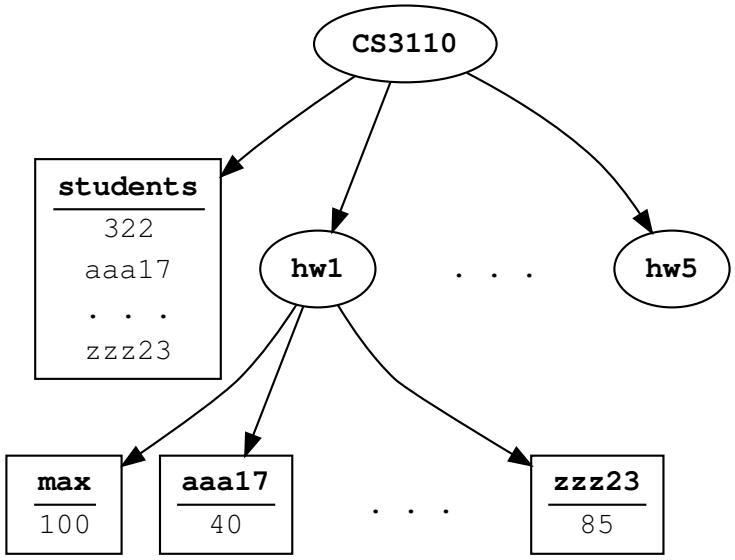



**Figure 5.1:** Example: file system fragment used to store course data.

The contributions of this chapter are:

- We present Transactional Forest, a declarative domain-specific language for processing ad hoc data in concurrent settings (Sections 5.3 and 5.4).
- We describe a prototype implementation of Transactional Forest as an embedded domain-specific language in OCaml and Python (Section 5.5).
- We prove that our design satisfies several formal properties, including round-tripping laws and serializability (Appendix B).

The next section motivates TxForest using the running example of the dissertation.

## 5.2 Example: Course Management System

This section motivates the design of TxForest using the simple running example introduced in Chapter 2. Recall the filestore fragment shown in Figure 5.1. The top-level

directory (**CS3110**) contains a file (**students**) and a set of sub-directories, one for each homework assignment (**hw1**–**hw5**). The **students** file gives the total number of enrolled students, followed by a list of their NetIDs (Cornell's unique identifier for its students and employees). Each homework directory has a file for each student that contains their grade on the assignment (*e.g.*, **aaa17**), as well as a special file (**max**) with the maximum possible score.

In our running example, we implemented a function to add students to our filestore. This requires the user to read and write the **students** file as well as to add a file to each of the **hw** directories. As a reminder, here is the Forest specification shown in Section 2.3:

```
hw = directory {
  max is "max" :: (pads pint);
  students is [student :: (pads student)
               | student <- matches RE "[a-z]+[0-9]+"]}

cs3110 = directory {
  studentList is "students" :: (pads studentsFile);
  hws is [name :: hw | name <- matches RE "hw[0-9]+"]}
```

The implementation of `add_student` from the same section is

```
let add_student ~student () =
  let (rep,md) = cs3110_load baseDir in
  Forest.exit_on_error md;
  if List.mem rep.studentList.pstudents student ~equal:String.equal
  then failwithf "add_student: Student %s already exists." student ()
  else
    add_student_to_filestore student (rep,md)
    |> cs3110_manifest
    |> Forest.exit_on_mani_error
    |> store
```

and has several important benefits over the OCaml code (Section 2.1): (1) the structure of the filestore is explicit in the specification and the code; (2) the use of types makes certain programming mistakes impossible, such as attempting to read a file at a missing path; and (3) any part of the filestore not conforming to the specification is automatically detected.

However, the Forest code does not support concurrency. Consider what happens if multiple members of the course staff try to add students concurrently. If they all read the **`students`** file before any write it, then we could easily be left with an inconsistent filestore because several students are present in the homeworks but not correctly 'registered' for the course. Other procedures, like renormalizations, produce more insidious inconsistencies, which are very difficult to detect, diagnose, and fix.

Further, it is unnecessary (and often infeasible) to load the entire filestore into memory—*e.g.*, suppose we only need to manipulate data for a single homework or an individual student. Chapter 3 dealt specifically with this problem; this chapter uses a different method to achieve the same result.

**Transactional Forest Implementation.** TxForest offers the same advantages as Forest, while dealing with issues related to concurrency and incrementality. The only added cost is a small shift in programming style—*i.e.*, navigating using a zipper.

The TxForest specification for our running example is identical to the Forest version. However, this surface-level specification is then translated to a core language (Section 5.3) that uses Huet's zipper internally and also provides transactional ACID guarantees. The TxForest code for the `add_student` function is different than the Forest version. Here is one possible implementation:

```
let add_student ~student z : (zipper,string) Result.t =
  let%bind l = get_student_names z in
  if List.mem l student ~equal:String.equal
  then mk_err "add_student: Student %s already exists." student
  else add_student_to_filestore student z
```

Note that the type of the function has changed so that it takes a zipper as an argument and returns a value in the result monad:

```
type ('a,'b) Result.t = Ok of 'a | Error of 'b
```

This result monad tracks the same sorts of errors seen in the Forest code—*e.g.* from malformed filestores but not from concurrency issues. The `let%bind` operator is a

shorthand for sequencing computation with the standard monadic bind operation. The `mk_err` function also allows users to thread their own errors through the monad.

Beyond this difference, the code is simple and clear; `add_student_to_filestore` is doing all of the work. As before, it is composed of two parts:

```
let add_student_to_filestore student z =
  add_student_to_studFile student z
  >>= add_student_to_every_hw student
```

The first part is rather uninteresting, but the second part utilizes several instructive constructs:

```
let add_student_to_every_hw student z =
  let%bind z = goto "hws" z in
  let add_student_to_hw z =
    down z >>= goto "students"
    >>= add_and_goto student
    >>= down >>= set_score Ungraded
    >>= up >>= up >>= up >>= up
  in
  Derived.map ~f:add_student_to_hw z
```

The `goto` function traverses the zipper—*e.g.*, `goto "hws" z` navigates to the directory node named `hws`, which is a comprehension. The `Derived.map` function at the bottom is a function derived from our core language (Section 5.3) that maps over a comprehension, performing a function at each node. The `add_student_to_hw` function then goes to the current homework and adds the student. First, it moves `down` the path specification (`name::hw`) to the homework directory. Then, it uses the *bind* operator (`>>=`) to thread the resulting zipper through the monad, and `goto` the `students` portion of the homework. It uses another derived function that adds the student to the comprehension and moves to that node. Then it walks `down` the path to the file node and updates the score. Finally, it moves `up` four times to get back to where it started, allowing `map` to correctly move to the next node and repeat the process.

Users need some way to construct a zipper to use the `add_student` function. The TxForest library provides functions called `run_txn` and `loop_txn`:

```
type txError = TxError | OpError of string
val run_txn : spec -> path -> (zipper -> ('a,string) Result.t) ->
              (unit -> ('a,txError) Result.t)
val loop_txn : spec -> path -> (zipper -> ('a,string) Result.t) ->
               (unit -> ('a,string) Result.t)
```

which might be used as follows:

```
match run_txn cs3110_spec path (add_student ~student:"jd753") () with
| Error TxError -> printf "Transaction aborted due to conflict"
| Error(OpError err) -> printf "Transaction aborted with error: %s" err
| Ok _ -> printf "Student jd753 added successfully"
```

The `run_txn` function takes a specification, an initial path, and a function from zippers to results and produces a thunk. When the thunk is forced, it constructs a zipper focused on the given path and runs the function. If this execution results in an error, then the outer computation produces an `OpError`. Otherwise, it attempts to commit the modifications produced during the computation. If the commit succeeds, then it returns the result of the function; otherwise it discards the results and returns a `TxError`. The `loop_txn` function is similar, but retries the transaction until there is no conflict or the input function produces an error.

TxForest guarantees that committed transactions are serializable—*i.e.*, the final file system will be equivalent to one produced by executing the committed transactions in some serial order. See Section 5.4 for the formal concurrent semantics and the serializability theorem. In our example, this means that no errors can occur due to adding multiple students simultaneously. Furthermore, TxForest automatically provides incrementality by only loading the data needed to traverse the zipper—an important property in larger filestores. Incremental Forest (Chapter 3) provides a similar facility, but requires explicit user annotations. Overall, TxForest provides incremental support for filestore applications in the presence of concurrency. The next two sections present the language in detail, develop an operational model, and establish its main formal properties.

## 5.3 Transactional Forest

This section presents TxForest in terms of a core calculus. We discuss the goals and high level design decisions for the language before formalizing the syntax, semantics, and several properties including round-tripping laws, equational identities, and consistency relations. Finally, we give a taste of the core calculus by using it to encode functions that would be useful for the course management example above. This section deals primarily with the single-threaded semantics, while the next section presents a concurrent model.

The main goals of this language are to allow practical processing of filestores for non-expert users. This leads to several requirements: (1) an intuitive way of specifying filestores [7]; (2) automatic, incremental processing, since filestores are typically large; (3) automatic concurrency control, as concurrency is both common and difficult to get right; and (4) transparency, since filestore interaction can be expensive and should therefore be explicit.

The zipper abstraction on which our language is based helps achieve our second and fourth requirements. Both of these requirements and concurrency are then further addressed by our locality-centered language design. The semantics of every command and expression only considers the locale around the focus node of the zipper. This means that every command can restrict its attention to a small part of the filestore which, along with the fact that data can be loaded as-required while traversing the zipper, gives incrementality. We believe that the combination of locality and explicit zipper traversal commands also gives transparency. In particular, the footprint of any command is largely predictable based on the filestore specification and current state. Predictability also simplifies tasks such as logging reads and writes, which is useful for concurrency control.

| | | | |
|---|---|---|---|
| Strings | | $u \in$ | $\Sigma^*$ |
| Integers | | $n \in$ | $\mathbb{Z}$ |
| Variables | | $x \in$ | *Var* |
| Values | | $v \in$ | *Val* |
| Environments | $E \in \mathit{Env}$ | : | $\mathit{Var} \mapsto \mathit{Val}$ |
| Paths | | $p ::=$ | $/ \mid p/u$ |
| Contents | | $C ::=$ | $\mathsf{Dir}\ \{\overline{u}\} \mid \mathsf{File}\ u$ |
| File Systems | | $fs$ : | $\mathit{Path} \mapsto \mathit{Content}$ |
| Contexts | | $ctxt$ : | $\mathit{Env} \times \mathit{Path} \times 2^{\mathit{Path}} \times \mathit{Zipper}$ |

**Figure 5.2:** Preliminaries

### 5.3.1 Syntax

We model a file system as a map from paths to file system contents, which are either directories (a set of their children's names) or files (strings). For a path $p$ and file system $fs$, we define $p \in fs \triangleq p \in \mathsf{dom}(fs)$. See Figure 5.2 for the metavariable conventions used in our formalization. We assume that all file systems are well-formed—*i.e.*, that they encode a tree, where each node is either a directory or a file with no children. We use Definition 2.3.1 from Chapter 2:

**Definition 2.3.1** (Well-Formedness)**.** A file system $fs$ is *well-formed* if and only if:

1. $fs(/) = \mathsf{Dir}\ \_$ (where / is the root node), and
2. $p/u \in fs \iff fs(p) = \mathsf{Dir}\ \ell \ \wedge\ u \in \ell$

In this definition, the notation _ indicates an irrelevant hole, which may be filled by any well-typed term. We use this convention throughout the chapter and dissertation.

In the previous section, we gave a flavor of the specifications one might write in TxForest. We wrote these specifications in our surface language, which compiles down to a core calculus, whose syntax is given in Figure 5.3. The core specifications are described fully below, but to first provide an intuition, we show the translation of a simplified version of the hw specification from Section 5.2:

| | | | |
|---|---|---|---|
| Specifications | $s \in \mathit{Spec}$ | ::= | $\mathit{File} \mid \mathit{Dir} \mid e :: s \mid \langle x{:}s_1, s_2 \rangle$ |
| | | $\mid$ | $[s \mid x \in e] \mid s? \mid P(e)$ |
| Zippers | $z$ | ::= | {`ancestor` : *Zipper* `option`; |
| | | | `left` : $(\mathit{Env} \times \mathit{Spec})$ `list`; |
| | | | `focus` : $(\mathit{Env} \times \mathit{Spec})$; |
| | | | `right` : $(\mathit{Env} \times \mathit{Spec})$ `list`} |
| Commands | $c$ | ::= | $\mathit{fc} \mid$ `Skip` $\mid c_1; c_2 \mid x := e$ |
| | | $\mid$ | `If` $b$ `Then` $c_1$ `Else` $c_2 \mid$ `While` $b$ `Do` $c$ |
| Forest Commands | $\mathit{fc}$ | ::= | $\mathit{fn} \mid \mathit{fu}$ |
| Forest Navigations | $\mathit{fn}$ | ::= | `Down` $\mid$ `Up` $\mid$ `Next` $\mid$ `Prev` |
| | | $\mid$ | `Into_Pair` $\mid$ `Into_Comp` $\mid$ `Into_Opt` $\mid$ `Out` |
| Forest Updates | $\mathit{fu}$ | ::= | `Store_File` $e \mid$ `Store_Dir` $e \mid$ `Create_Path` |
| Expressions | $e, b$ | ::= | $\mathit{fe} \mid v \mid x \mid e_1\ e_2 \mid \ldots$ |
| Forest Expressions | $\mathit{fe}$ | ::= | `Fetch_File` $\mid$ `Fetch_Dir` $\mid$ `Fetch_Path` |
| | | $\mid$ | `Fetch_Comp` $\mid$ `Fetch_Opt` $\mid$ `Fetch_Pred` |
| | | $\mid$ | `Run` $\mathit{fn}\ e \mid$ `Run` $\mathit{fe}\ e \mid$ `Verify` |
| Log Entries | $\mathit{le}$ | ::= | `Write_file` $C_1\ C_2\ p \mid$ `Read` $C\ p$ |
| | | $\mid$ | `Write_dir` $C_1\ C_2\ p$ |
| Logs | $\sigma$ | : | *LogEntry* `list` |
| Programs | $g$ | ::= | $(p, s, c)$ |

**Figure 5.3:** Main Syntax

```
directory {
  max is "max" :: file;
  students is [student :: file | s <- matches RE "[a-z]+[0-9]+"]}
```

becomes

$\langle \mathit{max}{:}$`"max"`$:: \mathit{File}, \langle \mathit{dir}{:}\mathit{Dir}, [s :: \mathit{File} \mid s \in e] \rangle \rangle$
`where` $e$ `= filter (Run Fetch_Dir` $\mathit{dir}$`)` `"[a-z]+[0-9]+"`

Directories become dependent pairs, allowing earlier parts of directories to be referenced by later parts. Comprehensions, which use regular expressions to query the file system, also turn into dependent pairs: The first component of the pair is a $\mathit{Dir}$. The second component fetches from and filters the first component using a regular expression.

Section 5.3.4 gives examples of functions written against this specification using the command language described below. We proceed by describing the syntax shown in Figure 5.3 in-depth.

Formally, a TxForest specification $s$ describes the shape and contents of a *filestore*, which is a structured subtree of a file system. Such specifications are almost identical to those in Forest [7]. Given the context of a current path $p$ and environment $E$, specifications can be understood as follows:

- *Files and Directories.* The $File$ and $Dir$ specifications describe filestores with a file and directory, respectively, at $p$.
- *Paths.* The $e :: s$ specification describes a filestore modeled by $s$ at the extension of $p$ by the value denoted by $e$ (evaluated in $E$).
- *Dependent Pairs.* The $\langle x{:}s_1, s_2 \rangle$ specification describes a filestore modeled by both $s_1$ and $s_2$. Additionally, $s_2$ may use the variable $x$ to refer to the portion of the filestore matched by $s_1$.
- *Comprehensions.* The $[s \mid x \in e]$ specification describes a filestore modeled by $s$ when $x$ is bound to any element in the set denoted by $e$ and is only productively used when $s$ depends on $x$.
- *Options.* The $s?$ specification describes a filestore where $p$ is either unmapped in the file system or modeled by $s$.
- *Predicates.* The $P(e)$ specification describes a filestore where boolean $e$ (evaluated in $E$) is `true`. This construct is typically used with dependent pairs.

Most specifications can be thought of as trees with as many children as they have sub-specifications. Comprehensions are the exception; we think of them as having as many children as there are elements in the set $e$.

To enable incremental and transactional manipulation of data contained in filestores, TxForest uses a zipper that is constructed from a specification. The zipper traverses the specification tree while keeping track of an environment that binds variables from dependent pairs and comprehensions. The zipper can be thought of as representing a tree along with the particular node of the tree that is in focus. We use the symbol `focus` to represents the focus node; `left` and `right` represent its siblings to the left and right respectively. The symbol `ancestor` tracks the focus node's ancestors by containing the value of the zipper before moving down to this depth of the tree. Key principles to keep in mind regarding zippers are that (1) the tree can be unfolded as it is traversed and (2) operations near the current node are fast, thus optimizing for locality.

To navigate a zipper, we use standard imperative (IMP) commands $c$ as well as special-purpose Forest Commands $fc$, divided into Forest Navigations $fn$ and Forest Updates $fu$. Navigation commands traverse the zipper, while Update commands modify the file system. Expressions are mostly standard and pure: they never modify the file system and only Forest Expressions query it. Forest Commands and Expressions will be described in greater detail in Section 5.3.2. To ensure serializability among multiple TxForest threads executing concurrently, we will maintain a log. An entry `Read` $C$ $p$ indicates that we have read $C$ at path $p$ while `Write_file` $C_1$ $C_2$ $p$ (respectively `Write_dir` $C_1$ $C_2$ $p$) indicates that we have written the file (respectively directory) $C_2$ to path $p$, where $C_1$ was before.

### 5.3.2 Semantics

Having defined the syntax, we now present the denotational semantics of TxForest. The semantics of IMP commands are standard and thus elided. We start by defining the semantics of a program:

$$[\![(p, s, c)]\!]_g\ fs \triangleq \texttt{project\_fs}\ ([\![c]\!]_c\ (\{\}, p, \{\}, \wr\{\}, s\wr)\ fs)$$

The denotation of a TxForest program is a function on file systems. We use the specification $s$ to construct a new zipper, seen in the figure using our zipper notation defined in Section 4.2 and recapitulated below. Then we execute the command $c$ using the denotation function for commands $[\![\cdot]\!]_c$. This function takes a context, which we construct using the zipper and the path $p$, and a file system $fs$ as arguments. The denotation function then produces a new context and file system, from which we project out the file system with `project_fs`. As a reminder, we recapitulate our zipper notation from Chapter 4:

**Definition 4.2.1** (Zipper Notation)**.** We define notation for constructing and deconstructing (*i.e.* pattern matching on) zippers. To construct a zipper we write:

$$\texttt{left} \leftharpoonup \wr \texttt{focus} \wr^{z} \rightharpoonup \texttt{right} \triangleq \{\texttt{ancestor} = \texttt{Some}(z); \texttt{left}; \texttt{focus}; \texttt{right}\}$$

where any of `ancestor`, `left`, and `right` can be omitted to denote a zipper with `ancestor` $=$ `None`, `left` $= []$, and `right` $= []$ respectively. For example:

$$\wr \texttt{focus} \wr \triangleq \{\texttt{ancestor} = \texttt{None}; \texttt{left} = []; \texttt{focus}; \texttt{right} = []\}$$

Likewise, to destruct a zipper we write:

$$\texttt{left} \leftharpoonup (\!|\texttt{focus}|\!)^{z} \rightharpoonup \texttt{right}$$

where any part can be omitted to ignore that portion of the zipper, but any included part must exist. For example, $z = (\!|\_|\!)^{z'} \iff z.\texttt{ancestor} = \texttt{Some}(z')$.
Finally, to construct a new zipper from an existing zipper and a focus node, we define:

$$z \textit{ with } f \triangleq \{z \texttt{ with focus} = f\}$$

The two key invariants that hold during the execution of any command are (1) that the file system remains well-formed (Definition 2.3.1); and (2) that if $p \in fs$ and $[\![fc]\!]_c\ (\_, p/u, \_, \_)\ fs = ((\_, p'/u', \_, \_), fs', \_)$ , then $p' \in fs'$. The first property states that

no command can make a well-formed file system ill-formed. The second states that, as we traverse the zipper, we maintain a connection to the real file system. It is important that only the parent of the current file system node is required to exist since this allows us to construct new portions of the filestore and handle the option specification. A central design choice that underpins the semantics is that each command acts *locally* on the current zipper and does not require further context. This makes the cost of the operation apparent and, as in Incremental Forest [4], facilitates partial loading and storing. These properties can be seen from Figure 5.4 which defines the semantics of Forest Commands.

Each rule of the figure defines the meaning of evaluating a command in a given context $(E, p, ps, z)$ and file system $fs$. The denotation function is partial, being undefined if none of the rules apply. Intuitively, a command is undefined when it is used on a malformed filestore with respect to its specification, or when it is ill-typed because it is used on an unexpected zipper state. Operationally, the semantics of each command can be understood as follows:

- `Down` and `Up` are duals: `Down` traverses the zipper into a path expression, simultaneously moving down in the file system, while `Up` does the reverse. Additionally, `Down` queries the file system, producing a `Read`.
- `Into` and `Out` are duals: `Into` traverses the zipper into its respective type of specification, while `Out` moves back out to the parent node. Additionally, their subexpressions may produce logs.

  For dependent pairs, we update the environment of the second child with a context constructed from the first specification.

  For comprehensions, the traversal requires the set denoted by $e$ to be non-empty, and maps it to a list of children with the same specification, but environments with different mappings for $x$, before moving to the first child.
- `Next` and `Prev` are duals: `Next` traverses the zipper to the right sibling and `Prev`

$$\frac{z = (\!|E_L, e :: s|\!) \qquad (u, \sigma) = [\![e]\!]_e\ (E_L, p, ps, z)\ fs \qquad \mathsf{Dir}\ \ell = fs(p) \qquad \sigma' = \sigma \cdot (\mathsf{Read}\ (\mathsf{Dir}\ \ell)\ p)}{[\![\mathtt{Down}]\!]_c\ (E, p, ps, z)\ fs = ((E, p/u, ps \cup (p/u), \wr E_L, s \int^z), fs, \sigma')}$$

$$\frac{z = (\!|\_|\!)^{z'} \qquad z' = (\!|\_, e :: s|\!)}{[\![\mathtt{Up}]\!]_c\ (E, p, ps, z)\ fs = ((E, \mathsf{pop}\ p, ps, z'), fs, \varepsilon)}$$

$$\frac{z = (\!|E_L, s?|\!)}{[\![\mathtt{Into\_Opt}]\!]_c\ (E, p, ps, z)\ fs = ((E, p, ps, \wr E_L, s \int^z), fs, \varepsilon)}$$

$$\frac{z = (\!|E_L, \langle x{:}s_1, s_2 \rangle|\!) \qquad ctxt = (E_L, p, ps, \wr E_L, s_1 \int)}{[\![\mathtt{Into\_Pair}]\!]_c\ (E, p, ps, z)\ fs = ((E, p, ps, \wr E_L, s_1 \int^z \rightharpoonup [(E_L[x \mapsto ctxt], s_2)]), fs, \varepsilon)}$$

$$\frac{z = (\!|E_L, [s \mid x \in e]|\!) \qquad (h \cdot t, \sigma) = [\![e]\!]_e\ (E_L, p, ps, z)\ fs \qquad r = \mathsf{map}\ (\lambda u.\ (E_L[x \mapsto u], s))\ t}{[\![\mathtt{Into\_Comp}]\!]_c\ (E, p, ps, z)\ fs = ((E, p, ps, \wr E_L[x \mapsto h], s \int^z \rightharpoonup r), fs, \sigma)}$$

$$\frac{z = (\!|\_|\!)^{z'} \qquad z' \neq (\!|\_, e :: s|\!)}{[\![\mathtt{Out}]\!]_c\ (E, p, ps, z)\ fs = ((E, p, ps, z'), fs, \varepsilon)}$$

$$\frac{z = l \leftharpoonup (\!|f'|\!)^{z'} \rightharpoonup (f \cdot r)}{[\![\mathtt{Next}]\!]_c\ (E, p, ps, z)\ fs = ((E, p, ps, (f' \cdot l) \leftharpoonup \wr f \int^{z'} \rightharpoonup r), fs, \varepsilon)}$$

$$\frac{z = (f \cdot l) \leftharpoonup (\!|f'|\!)^{z'} \rightharpoonup r}{[\![\mathtt{Prev}]\!]_c\ (E, p, ps, z)\ fs = ((E, p, ps, l \leftharpoonup \wr f \int^{z'} \rightharpoonup (f' \cdot r)), fs, \varepsilon)}$$

$$\frac{z = (\!|\_, File|\!) \qquad (u, \sigma) = [\![e]\!]_e\ (E, p, ps, z)\ fs \qquad (fs', \sigma') = \mathtt{make\_file}\ fs\ p\ u}{[\![\mathtt{Store\_File}\ e]\!]_c\ (E, p, ps, z)\ fs = ((E, p, ps, z), fs', \sigma \cdot \sigma')}$$

$$\frac{z = (\!|\_, Dir|\!) \qquad (\ell, \sigma) = [\![e]\!]_e\ (E, p, ps, z)\ fs \qquad (fs', \sigma') = \mathtt{make\_directory}\ fs\ p\ \ell}{[\![\mathtt{Store\_Dir}\ e]\!]_c\ (E, p, ps, z)\ fs = ((E, p, ps, z), fs', \sigma \cdot \sigma')}$$

$$\frac{z = (\!|E_L, e :: s|\!) \qquad (u, \sigma) = [\![e]\!]_e\ (E_L, p, ps, z)\ fs \qquad (fs', \sigma') = \mathtt{create}\ fs\ p/u \qquad \sigma'' = \sigma \cdot (\mathsf{Read}\ fs(p)\ p) \cdot \sigma'}{[\![\mathtt{Create\_Path}]\!]_c\ (E, p, ps, z)\ fs = ((E, p, ps, z), fs', \sigma'')}$$

**Figure 5.4:** *fc* Command Semantics

moves to the left sibling.

- `Store_File` $e$, `Store_Dir` $e$, and `Create_Path` all update the file system, leaving the zipper untouched. The definitions of the helper functions that they use can be found in Figure 5.6. All of them maintain a well-formed file system and produce logs recording their effects.

  For `Store_File` $e$, $e$ must evaluate to a string $u$, after which the command turns the current file system node into a file containing $u$.

  For `Store_Dir` $e$, $e$ must evaluate to a string set $\ell$, after which the command turns the current file system node into a directory containing that set. If the node is already a directory containing $\ell'$, then any children in $\ell' \setminus \ell$ are removed, any children in $\ell \setminus \ell'$ are added (as empty files), and any children in $\ell \cap \ell'$ are untouched.

  For `Create_Path`, the current node is turned into a directory containing the path to which the path expression points. The operation is idempotent and does the minimal work required: If the current node is already a directory, then the path is added. If the path was already there, then `Create_Path` is a no-op, otherwise it will map to an empty file.

We have covered the semantics of all of the Forest Commands, but their subexpressions remain. The semantics of non-standard expressions is given in Figure 5.5. The interpretation of each rule is the same as for commands. There is one `Fetch` expression per specification except for pairs, which have no useful information available locally. Since a pair is defined in terms of its sub-specifications, we must navigate to them before fetching information from them. This design avoids incurring the cost of eagerly loading a large filestore.

Fetching a file returns the string contained by the file at the current path. For a directory, we get the names of its children. Both of these log `Reads` since they inspect the file system. For a path specification, the only locally available information is the actual

$$\frac{z = (\!|\_, \mathit{File}|\!) \qquad \mathsf{File}\ u = fs(p)}{[\![\texttt{Fetch\_File}]\!]_e\ (E, p, ps, z)\ fs = (u, [\mathsf{Read}\ (\mathsf{File}\ u)\ p])}$$

$$\frac{z = (\!|\_, \mathit{Dir}|\!) \qquad \mathsf{Dir}\ \ell = fs(p)}{[\![\texttt{Fetch\_Dir}]\!]_e\ (E, p, ps, z)\ fs = (\ell, [\mathsf{Read}\ (\mathsf{Dir}\ \ell)\ p])}$$

$$\frac{z = (\!|E_L, e :: s|\!)}{[\![\texttt{Fetch\_Path}]\!]_e\ (E, p, ps, z)\ fs = [\![e]\!]_e\ (E_L, p, ps, z)\ fs}$$

$$\frac{z = (\!|E_L, [s \mid x \in e]|\!)}{[\![\texttt{Fetch\_Comp}]\!]_e\ (E, p, ps, z)\ fs = [\![e]\!]_e\ (E_L, p, ps, z)\ fs}$$

$$\frac{z = (\!|\_, s?|\!)}{[\![\texttt{Fetch\_Opt}]\!]_e\ (E, p, ps, z)\ fs = (p \in fs, [\mathsf{Read}\ fs(p)\ p])}$$

$$\frac{z = (\!|E_L, P(e)|\!)}{[\![\texttt{Fetch\_Pred}]\!]_e\ (E, p, ps, z)\ fs = [\![e]\!]_e\ (E_L, p, ps, z)\ fs}$$

$$\frac{(\mathit{ctxt}', \sigma') = [\![e]\!]_e\ (E, p, ps, z)\ fs \qquad (\mathit{ctxt}, fs, \sigma) = [\![\mathit{fn}]\!]_c\ \mathit{ctxt}'\ fs}{[\![\mathsf{Run}\ \mathit{fn}\ e]\!]_e\ (E, p, ps, z)\ fs = (\mathit{ctxt}, \sigma' \cdot \sigma)}$$

$$\frac{(\mathit{ctxt}, \sigma') = [\![e]\!]_e\ (E, p, ps, z)\ fs \qquad (v, \sigma) = [\![\mathit{fe}]\!]_e\ \mathit{ctxt}\ fs}{[\![\mathsf{Run}\ \mathit{fe}\ e]\!]_e\ (E, p, ps, z)\ fs = (v, \sigma' \cdot \sigma)}$$

$$\frac{(p', z') = \texttt{goto\_root}\ (E, p, ps, z)\ fs}{[\![\texttt{Verify}]\!]_e\ (E, p, ps, z)\ fs = \texttt{PConsistent}\ (p', ps, z')\ fs}$$

**Figure 5.5:** Expression Semantics

path. For a comprehension, we return the set $e$. For an option, we determine whether the current path is in the file system and log a `Read` regardless. Finally, for a predicate, we determine if its condition holds.

There are two `Run` expressions. Subexpression $e$ must evaluate to a context. These can only come from a dependent pair, which means that `Run` can only occur as a subexpression of specifications. We utilize them by performing traversals (`Run` $\mathit{fn}\ e$) and evaluating Forest expressions (`Run` $\mathit{fe}\ e$) in the input context. For example, a filestore defined by a file `index.txt` and a set of files listed in that index could be described as

```
make_file : File system → Path → Σ* → File system × Log
make_directory : File system → Path → 2^Σ* → File system × Log
create : File system → Path → Σ* → File system × Log
close_fs : File system → File system
```

```
make_file fs p u ≜
  let (fs', σ') = create fs p in
  let σ = σ' · (Write_file fs'(p) (File u) p) in
  (close_fs (fs'[p ↦ File u]), σ)


make_directory fs p ℓ ≜
  let (fs', σ') = create fs p in
  let σ = σ' · (Write_dir fs'(p) (Dir ℓ) p) in
  (close_fs (fs'[p ↦ Dir ℓ]), σ)


create fs p/u ≜
  match fs(p) with
  | ⊥ →
    let (fs', σ') = create fs p in
    let σ = σ' · (Write_dir (File ε) (Dir {u}) p) in
    (close_fs (fs'[p ↦ Dir {u}]), σ)
  | File u' →
    (close_fs (fs[p ↦ Dir {u}]), [Write_dir (File u') (Dir {u}) p])
  | Dir ℓ when u ∉ ℓ →
    (close_fs (fs[p ↦ Dir (ℓ ∪ {u})]), [Write_dir (Dir ℓ) (Dir (ℓ ∪ {u})) p])
  | Dir ℓ when u ∈ ℓ → (fs, [])


close_fs fs ≜ close_at fs /
  where close_at fs p ≜
    match fs(p) with
    | Dir ℓ →
      let ℓ' = {p/u | u ∈ ℓ} in
      let fs' = fold fs ℓ' close_at in
      let ℓ'' = {p' ∈ fs | subpath p' p ∧ ∀p'' ∈ ℓ'. ¬subpath p' p''} in
      let fs'' = fold fs' ℓ'' (λfs p'. fs[p' ↦ ⊥]) in
      fs''[p ↦ Dir ℓ]
    | File u →
      let fs' = fold fs {p' ∈ fs | subpath p' p} (λfs p'. fs[p' ↦ ⊥]) in
      fs'[p ↦ File u]
    | ⊥ → fs[p ↦ File ε]
```

**Figure 5.6:** TxForest Helper Functions

follows:

$\langle index$:`"index.txt"` $:: File, [x :: File \mid x \in e]\rangle$
`where` $e$ `= lines_of (Run Fetch_File (Run Down` $index$`))`

where `lines_of` maps a string to a string set by splitting it by lines.

Finally, `Verify` checks the partial consistency of the traversed part of the filestore—whether it conforms to our specification. Unfortunately, checking an entire filestore, even incrementally, can be very expensive and, often, we have only performed some local changes and thus do not need the full check. Partial consistency is a compromise wherein we check only the portions of the filestore that we have traversed, as denoted by the path set. This ensures that the cost of the check is proportional to the cost of the operations we have already run. Partial consistency is formally defined in the next subsection, which among other properties, details the connection between partial and full consistency.

### 5.3.3 Properties

This section establishes properties of the TxForest core calculus: consistency and partial consistency, equational identities on commands, and round-tripping laws.

The formal definition of partial consistency is given in Figure 5.7. Intuitively, full consistency (`Consistent`) captures whether a filestore conforms to its specification. For example, the file system $fs$ at $p$ conforms to $File$ if and only if $fs(p) =$ `File _` and to $e :: s$ if $e$ evaluates to $u$ and $fs$ at $p/u$ conforms to $s$. Partial consistency (`PConsistent`) then checks partial conformance (*i.e.* does the filestore conform to part of its specification). `PConsistent` returns two booleans (and a log), the first describing whether the input filestore is consistent with the input specification and the second detailing whether that consistency is total or partial. The definition of full consistency is similar to partial consistency, except that there are no conditions and the path set is ignored. The properties

$$\frac{p \notin ps}{\texttt{PConsistent}\ (p, ps, z)\ fs = ((\texttt{true}, \texttt{false}), \varepsilon)}$$

$$\frac{p \in ps}{\texttt{PConsistent}\ (p, ps, (\!|(E, File)|\!)\ \texttt{as}\ z)\ fs = ((fs(p) = \mathsf{File}\ \_, \texttt{true}), [\mathsf{Read}\ fs(p)\ p])}$$

$$\frac{p \in ps}{\texttt{PConsistent}\ (p, ps, (\!|(E, Dir)|\!)\ \texttt{as}\ z)\ fs = ((fs(p) = \mathsf{Dir}\ \_, \texttt{true}), [\mathsf{Read}\ fs(p)\ p])}$$

$$\frac{\begin{array}{c} p \in ps \qquad (u, \sigma) = [\![e]\!]_e\ (E, p, ps, z)\ fs \\ ret = ((fs(p) = \mathsf{Dir}\ \_, \texttt{true}), \sigma \cdot (\mathsf{Read}\ fs(p)\ p)) \\ ret' = ret\ \wedge_\sigma\ \texttt{PConsistent}\ (p/u, ps, \wr(E, s)\wr^z)\ fs \end{array}}{\texttt{PConsistent}\ (p, ps, (\!|(E, e :: s)|\!)\ \texttt{as}\ z)\ fs = ret'}$$

$$\frac{\begin{array}{c} p \in ps \qquad ctxt = (E, p, ps, \wr(E, s_1)\wr) \qquad E' = E[x \mapsto ctxt] \\ ret = \texttt{PConsistent}\ (p, ps, \wr(E, s_1)\wr^z \rightharpoonup [(E', s_2)])\ fs \\ ret' = ret\ \wedge_\sigma\ \texttt{PConsistent}\ (p, ps, [(E, s_1)] \leftharpoonup \wr(E', s_2)\wr^z)\ fs \end{array}}{\texttt{PConsistent}\ (p, ps, (\!|(E, \langle x{:}s_1, s_2\rangle)|\!)\ \texttt{as}\ z)\ fs = ret'}$$

$$\frac{\begin{array}{c} p \in ps \qquad (\ell, \sigma') = [\![e]\!]_e\ (E, p, ps, z)\ fs \\ ((b_1, b_2), \sigma) = \bigwedge_{v \in \ell} \texttt{PConsistent}\ (p, ps, \wr(E[x \mapsto v], s)\wr^z)\ fs \end{array}}{\texttt{PConsistent}\ (p, ps, (\!|(E, [s \mid x \in e])|\!)\ \texttt{as}\ z)\ fs = ((b_1, b_2), \sigma' \cdot \sigma)}$$

$$\frac{p \in ps \qquad ret = ((p \notin fs, \texttt{true}), [\mathsf{Read}\ fs(p)\ p])\ \vee_\sigma\ \texttt{PConsistent}\ (p, ps, \wr(E, s)\wr^z)\ fs}{\texttt{PConsistent}\ (p, ps, (\!|(E, s?)|\!)\ \texttt{as}\ z)\ fs = ret}$$

$$\frac{p \in ps \qquad (b, \sigma) = [\![e]\!]_e\ (E, p, ps, z)\ fs}{\texttt{PConsistent}\ (p, ps, (\!|(E, P(e))|\!)\ \texttt{as}\ z)\ fs = ((b, \texttt{true}), \sigma)}$$

$$\begin{array}{lcl}
((\texttt{false}, _), \sigma)\ \wedge_\sigma\ _ & \triangleq & ((\texttt{false}, \texttt{false}), \sigma) \\
((b_1, b_2), \sigma)\ \wedge_\sigma\ ((b_1', b_2'), \sigma') & \triangleq & ((b_1\ \wedge\ b_1', b_2\ \wedge\ b_2'), \sigma \cdot \sigma') \\
((\texttt{true}, \texttt{true}), \sigma)\ \vee_\sigma\ _ & \triangleq & ((\texttt{true}, \texttt{true}), \sigma) \\
((b_1, b_2), \sigma)\ \vee_\sigma\ ((b_1', b_2'), \sigma') & \triangleq & ((b_1\ \vee\ b_1', b_2\ \vee\ b_2'), \sigma \cdot \sigma') \\
\\
\texttt{complete?}\ ((_, b), _) & \triangleq & b \\
\texttt{consistent?}\ ((b, _), _) & \triangleq & b \\
\\
\texttt{Cover}\ (p, ps, z)\ fs & :\Longleftrightarrow & \texttt{complete?}\ (\texttt{PConsistent}\ (p, ps, z)\ fs)
\end{array}$$

**Figure 5.7:** Partial Consistency and Cover

below describe the relationship between partial consistency and full consistency. Their proofs can be found in Appendix B.1.2.

**Theorem 5.3.1.** *Consistency implies partial consistency:*

$$\forall ps.\ \texttt{consistent?}\ (\texttt{Consistent}\ (p, ps, z)\ fs) \implies$$
$$\texttt{consistent?}\ (\texttt{PConsistent}\ (p, ps, z)\ fs)$$

**Theorem 5.3.2.** *Partial Consistency is monotonic w.r.t. the path set:*

$$\forall ps_1, ps_2.\ ps_2 \subseteq ps_1 \implies$$
$$\texttt{consistent?}\ (\texttt{PConsistent}\ (p, ps_1, z)\ fs) \implies$$
$$\texttt{consistent?}\ (\texttt{PConsistent}\ (p, ps_2, z)\ fs)$$
$$\wedge\ \texttt{complete?}\ (\texttt{PConsistent}\ (p, ps_2, z)\ fs) \implies$$
$$\texttt{complete?}\ (\texttt{PConsistent}\ (p, ps_1, z)\ fs)$$

Theorem 5.3.2 says that if a filestore defined by $z$ is partially consistent with respect to $ps_1$, then it will also be partially consistent with respect to any path set $ps_2$ that is a subset of $ps_1$. Conversely, if partial consistency with respect to $ps_2$ is total, or complete, then as is partial consistency with respect to $ps_1$.

**Theorem 5.3.3.** *Given a zipper $z$ and a path set $ps'$ that covers the entirety of $z$, partial consistency holds iff full consistency holds:*

$$\forall ps, ps'.\ \texttt{Cover}\ (p, ps', z)\ fs\ \wedge\ ps' \subseteq ps \implies$$
$$\texttt{consistent?}\ (\texttt{Consistent}\ (p, ps, z)\ fs) \iff$$
$$\texttt{consistent?}\ (\texttt{PConsistent}\ (p, ps, z)\ fs)$$

Theorem 5.3.3 says that if the path set $ps$ is a superset of one that covers the entire filestore $ps'$, as defined in Figure 5.7, then the filestore is totally consistent exactly when it is partially consistent with respect to $ps$. Intuitively, if a path set covers a filestore then we can never encounter a path outside of the path set while traversing the zipper.

Other properties of the language include identities of the form $[\![\texttt{Down}; \texttt{Up}]\!]_c \equiv [\![\texttt{Skip}]\!]_c$ where $\equiv$ denotes equivalence modulo log, when defined. That is, either $[\![\texttt{Down}; \texttt{Up}]\!]_c$ is undefined, or it has the same action as $[\![\texttt{Skip}]\!]_c$, ignoring logging. Additionally, we have proven round-tripping laws in the style of lenses [12] stating, for example, that storing just loaded data is equivalent to `Skip`. Further identities and formal statements of these laws can be found in Appendix B.1.2.

### 5.3.4 Examples

This subsection details the core calculus encodings of a few useful functions for interfacing with the course management system introduced in Section 5.2. The goal is to build an intuition for the language and how to program with the zipper abstraction. In practice, a higher-level language would compile to this core calculus.

For these examples, we assume that `in` variables contain input arguments at the start of each function and that `out` variables should contain the output of the function, if any, at the end. Additionally, all examples are written against the same single-homework specification that we saw earlier, in both our higher-level description language and in the core calculus:

```
directory {
  max is "max" :: file;
  students is [student :: file | s <- matches RE "[a-z]+[0-9]+"]}
```

that is,

$\langle max$:"max":: $File, \langle dir$:$Dir, [s :: File \mid s \in e]\rangle\rangle$
where $e$ = filter (Run Fetch_Dir $dir$) "[a-z]+[0-9]+"

That said, we proceed to code simple primitive functions for getting and setting the score of a single student and for adding a student, a fold function over path comprehensions, and finally a function for getting the average score of all students for a single homework.

```
getScore := λ(). to_int Fetch_File
setScore ≜ Store_File (of_int in)
```

In `getScore` and `setScore`, we assume that the zipper is already at a student. `getScore`, which we can define as an expression in the language, takes a unit input and fetches the current file, converting the string to an integer. `setScore`, like the rest of our examples, is instead a metavariable representing a particular command. This command converts $in$ to a string before storing it as a file.

```
addStudent ≜
  Into_Pair; Next; Into_Pair;      # Go to dir
  Store_Dir (Fetch_Dir ∪ {in}); # Add in to the directory
  Out; Prev; Out                   # Return
```

In `addStudent`, we start from the root of the filestore and navigate to the first component of the internal pair. We then fetch the names of the current files in the directory before adding $in$ and storing it back. Finally, we return to the root.

```
fold ≜
  num := length Fetch_Comp; Into_Comp;
  While num > 0 Do
    Down;                    # Enter path
    inAcc := inF inAcc;      # Execute function and update accumulator
    Up; Next; num := num − 1 # Go to next element
  Out;
  out := inAcc
```

In `fold`, the zipper should start at a comprehension whose subspecification is a path expression. We take two inputs: $inAcc$, which is the initial accumulator value, and $inF$, which is a function that produces a new accumulator from the old one. The code for `fold` starts by getting the number of elements in the comprehension and then traverses the elements one by one, calling $inF$ to update the accumulator at each element.

Finally, `getAvg` computes the average score across all students:

```
getAvg ≜
  Into_Pair; Next; Into_Pair; Next;
  number := length Fetch_Comp;
  inAcc := 0;
  inF := λx. getScore () + x;
  fold;
  Prev; Out; Prev; Out;
  out := out / number
```

| | | |
|---|---|---|
| $ts \in$ | *Timestamp* | Timestamps |
| $GL \in$ | *TSLog* | Timestamped Logs |
| $td \in$ | *Thread* | $\triangleq$ *Context* $\times$ *File system* $\times$ *Command* |
| $TxS \in$ | *TxState* | $\triangleq$ *Command* $\times$ *Timestamp* $\times$ *Log* |
| $t \in$ | *Transaction* | $\triangleq$ *Thread* $\times$ *TxState* |
| $T \in$ | *Transaction Pool* | $\triangleq$ *Transaction* $Bag$ |

**Figure 5.8:** Global Semantics Additional Syntax

It starts at the root of the filestore and navigates to the comprehension. Next, it stores the number of students in $number$, sets $inAcc$ to $0$ and constructs $inF$, which gets the score of the current student and adds it to its argument. Then it folds, returns to the root of the filestore, and finally divides the result of the fold ($out$) by the number of students to obtain the final result.

## 5.4 Concurrency Control

This section introduces the global semantics of Transactional Forest, using both a denotational semantics to concisely capture a serial semantics and an operational semantics to capture thread interleavings and concurrency. We also state a serializability theorem that relates the two semantics.

Figure 5.8 lists the additional syntax used in this section. Timestamped logs are the logs of the global semantics. They are identical to local logs except that each entry also contains a timestamp indicating when it was written to the log.

Each *Thread* is captured by its local context which, along with its transactional state *TxState*, denotes a *Transaction*. The transactional state has 3 parts: (1) the command the transaction is executing; (2) the time when the transaction started; and (3) the transaction-local log recorded so far.

Our global denotational semantics is defined as follows:

$$[\![((ctxt, \_, c), \_)]\!]_G\ fs \triangleq \texttt{project\_fs}\ ([\![c]\!]_c\ ctxt\ fs)$$

$$[\![\ell]\!]_{\mathbb{G}}\ fs \triangleq \texttt{fold}\ fs\ \ell\ [\![\cdot]\!]_G$$

The denotation of one or more transactions is a function on file systems. For a single transaction, it is the denotation of the command with the encapsulated context, except for the file system that is replaced by the input. For a list of transactions, it is the result of applying the local denotation function in serial order. Note that the denotation of a transaction is precisely the denotation of a program, $[\![\cdot]\!]_g$, which can be lifted to multiple programs by folding. The key point about this semantics is that there is no interleaving of transactions. By definition, transactions run sequentially. While this ensures serializability, it also does not allow concurrency.

We will instead use an operational semantics that more easily models transaction interleaving and prove that it is equivalent to the denotational semantics. First, we introduce an operational semantics for local commands. This semantics is standard for IMP commands, but for Forest Commands, it uses the denotational semantics, considering each a single atomic step, as seen below:

$$\frac{((E', p', ps', z'), fs', \sigma) = [\![fc]\!]_c\ (E, p, ps, z)\ fs}{\langle (E, p, ps, z), fs, fc\rangle \xrightarrow{\sigma}_L \langle (E', p', ps', z'), fs', \texttt{Skip}\rangle}$$

Next, we can construct the global operational semantics, as seen in Figure 5.9. The global stepping relation is between two global contexts that have three parts: a global file system, a global log, and a transaction pool—*i.e.* a bag of transactions.

There are only three actions that the global semantics can take:

1. A transaction can step in the local semantics and append the resulting log.
2. A transaction that is done, and does not conflict with previously committed transactions, can commit. It must check that none of its operations conflicted with those

$$\frac{\langle td\rangle \xrightarrow{\sigma'}_L \langle td'\rangle}{\langle FS, GL, \{(td, (cs, ts, \sigma))\} \uplus T\rangle \to_G \langle FS, GL, \{(td', (cs, ts, \sigma \cdot \sigma'))\} \uplus T\rangle}$$

$$\frac{\texttt{is\_Done?}\ td \qquad \texttt{check\_log}\ GL\ \sigma\ ts \qquad FS' = \texttt{merge}\ FS\ \sigma \qquad GL' = GL \cdot (\texttt{add\_ts}\ \texttt{fresh\_ts}\ \sigma)}{\langle FS, GL, \{(td, (cs, ts, \sigma))\} \uplus T\rangle \to_G \langle FS', GL', T\rangle}$$

$$\frac{\texttt{is\_Done?}\ td \qquad \neg(\texttt{check\_log}\ GL\ \sigma\ ts) \qquad ts' = \texttt{fresh\_ts} \qquad (z', p') = \texttt{goto\_root}\ (E, p, ps, z)\ fs \qquad td' = ((\{\}, p', \{\}, z'), FS, cs)}{\langle FS, GL, \{(td, (cs, ts, \sigma))\} \uplus T\rangle \to_G \langle FS, GL, \{(td', (cs, ts', []))\} \uplus T\rangle}$$

**Figure 5.9:** Global Operational Semantics

$$\begin{aligned}
\texttt{merge}\ FS\ \sigma &\triangleq \texttt{fold}\ FS\ \sigma\ \texttt{update}\\
\texttt{update}\ fs\ (\texttt{Read}\ C\ p) &\triangleq fs\\
\texttt{update}\ fs\ (\texttt{Write_file}\ _\ C\ p) &\triangleq \texttt{close_fs}\ (fs[p \mapsto C])\\
\texttt{update}\ fs\ (\texttt{Write_dir}\ _\ C\ p) &\triangleq \texttt{close_fs}\ (fs[p \mapsto C])\\
\texttt{check_log}\ GL\ \sigma\ ts &\triangleq \forall p' \in \texttt{extract_paths}\ \sigma.\ \forall (ts', le) \in GL.\\
&\qquad ts > ts' \ \vee\ \neg(\texttt{conflict_path}\ p'\ le)\\
\texttt{conflict_path}\ p'\ (\texttt{Read}\ _\ p) &\triangleq \texttt{false}\\
\texttt{conflict_path}\ p'\ (\texttt{Write_file}\ _\ _\ p) &\triangleq \texttt{subpath}\ p'\ p\\
\texttt{conflict_path}\ p'\ (\texttt{Write_dir}\ _\ _\ p) &\triangleq \texttt{subpath}\ p'\ p\\
\texttt{extract_paths}\ [] &\triangleq \{\}\\
\texttt{extract_paths}\ ((\texttt{Read}\ _\ p) \cdot tl) &\triangleq \{p\} \cup (\texttt{extract_paths}\ tl)\\
\texttt{extract_paths}\ ((\texttt{Write_file}\ _\ _\ p) \cdot tl) &\triangleq \{p\} \cup (\texttt{extract_paths}\ tl)\\
\texttt{extract_paths}\ ((\texttt{Write_dir}\ _\ _\ p) \cdot tl) &\triangleq \{p\} \cup (\texttt{extract_paths}\ tl)
\end{aligned}$$

**Figure 5.10:** `merge` and `check_log`

committed since its start. Conflicts occur when the transaction read stale data. To commit, the transaction will update the global file system according to any writes performed. Finally, the transaction will leave the transaction pool. The definitions of `check_log` and `merge` can be found in Figure 5.10.

3. A transaction that is done but conflicts with previously committed transactions cannot commit; it instead must restart. It restarts by getting a fresh timestamp and resetting its log and local context.

In the operational semantics, transaction steps can be interleaved arbitrarily, but changes will get rolled back in case of a conflict. Furthermore, while Forest Commands are treated as atomic for simplicity they could also be modeled at finer granularity without affecting our results.

With a global semantics where transactions are run concurrently, we now aim to prove that our semantics guarantees serializability. The theorem below captures this property by connecting the operational and denotational semantics:

**Theorem 5.4.1** (Serializability)**.** *Let $FS, FS'$ be file systems, $GL, GL'$ be global logs, and $T$ a transaction pool such that $\forall t \in T.\ \texttt{initial}\ FS\ t$. Then:*

$$\langle FS, GL, T\rangle \rightarrow_G^* \langle FS', GL', \{\}\rangle \implies \exists \ell \in Perm(T).\ [\![\ell]\!]_{\mathbb{G}}\ FS = FS'$$

*where $\rightarrow_G^*$ is the reflexive, transitive closure of $\rightarrow_G$.*

The serializability theorem states that given a starting file system and a transaction pool of starting transactions, if the global operational semantics commits them all, then there is some ordering of these transactions for which the global denotational semantics will produce the same resulting file system. Note that although not required by the theorem, the commit order is one such ordering. Additionally, though not explicitly stated, it is easy to see that any serial schedule that is in the domain of the denotation function is realizable by the operational semantics. See Appendix B.1.1 for the proof.

The prototype system described in the next section implements the local semantics from the previous section along with this global semantics, reducing the burden of writing correct concurrent applications.

## 5.5 Implementation

This section describes our prototype implementation of Transactional Forest as an embedded domain-specific language in OCaml. Our prototype comprises 6089 lines of code (excluding blank lines and comments) and encodes Forest's features as a PPX syntax extension. We have also implemented a prototype in Python, which provides most of the features described below.

We have implemented a simple course management system similar to the running example from Section 5.2. Beyond adding students, the system has several additional facilities, including renormalizing grades, computing various statistics about students or homeworks and changing rubrics while automatically updating student grades accordingly. The most interesting piece of the example is based on our experience with a professional grading system that uses a queue from which graders can get new problems to grade. Unfortunately, this system did not adequately employ concurrency control, resulting in duplicated work. Using TxForest, we implemented a simple grading queue where graders can add and retrieve problems, which does not suffer from such concurrency issues.

The embedded language in our prototype implementation implements almost precisely the language seen in Section 5.3. Additionally, we provide a surface syntax (as seen in Section 5.2 and papers on the earlier versions of Forest [4, 7]) for specifications that compiles down to the core calculus seen in Section 5.3. This specification can then be turned into a zipper by initiating a transaction. The majority of the commands and expressions seen in the core semantics are exposed as functions in a library. Additionally, there is a more ad hoc surface command language that resembles the surface syntax and parallels the behavior of the core language. Finally, the global semantics looks slightly different compared to in Section 5.4, though this should not affect users and the minor

variant has been proven correct. We provide a simple shell for interacting with filestores, which makes it significantly easier to force conflicts and test the concurrent semantics.

## 5.6 Conclusion

This chapter has presented the design, syntax, and semantics of Transactional Forest, a domain-specific language for incrementally processing ad hoc data in concurrent applications. TxForest aims to provide an easier and less error-prone approach to modeling and interacting with a structured subset of a file system, which we call a *filestore*. We achieve this by using Huet's Zippers [23] as the core abstraction. This traversal-based structure naturally lends itself to incrementality and a simple, efficient, logging scheme that supports optimistic concurrency control. We provide a core language with a formal syntax and semantics based on zipper traversal, both for local, single-threaded applications, and for a global view with arbitrarily many Forest processes. We prove that this global view enforces serializability between threads, that is, the resulting effect on the file system of any set of concurrent threads is the same as if they had run in some serial order. Our OCaml and Python prototypes provides a surface language mirroring Forest [7] and a library of functions for manipulating the filestore.

While Transactional Forest offers significant benefits in managing concurrent filestores, it is unable to offer guarantees with respect to other (*i.e.* non-TxForest) concurrent processes on the file system. Additionally, the semantics and proofs are predicated on the correctness of the file system interface, but they unfortunately tend to be informally specified. The next chapter seeks to resolve these issues by designing a new transactional file system upon which Transactional Forest can run.

Chapter 6

# The Zipper File System:

# A Zipper-based Transactional File System

***Acknowledgments.*** *The content of this chapter was developed in collaboration with Nate Foster and Katie Mancini from Cornell University and Kathleen Fisher from Tufts University.*

## Brief Summary

This chapter introduces the Zipper File System (ZFS), a transactional file system with a novel underlying abstraction and a formal semantics describing its behavior. This file system is based on Kiselyov's ZFS design [27]. It uses a zipper abstraction to represent both the tree structure and the working path of a file system. We provide a denotational semantics to describe the operation of ZFS, both at the local and global level. We design an operational global semantics, which allows thread interleavings, and we prove that it provides serializable transactions among arbitrary concurrent processes by relating it to the denotational semantics. We also provide a translation from POSIX into ZFS to allow standard POSIX applications to benefit from the serializable transactions that ZFS supports. We also formalize a subset of POSIX denotationally in order to support this translation, where the formalization can be seen as a contribution of its own. Finally, we

have implemented a prototype of ZFS in OCaml.

## 6.1 Introduction

Transactional Forest (from the previous chapter) is a domain-specific language that lets users incrementally process ad hoc data in concurrent applications. TxForest leverages zippers to provide a formal semantics for both the local (single-threaded) and global (concurrent) operation of applications that interface with TxForest. We prove that the global semantics provides serializability among concurrent, TxForest threads.

TxForest suffers from two weaknesses: (1) it offers no guarantees with respect to concurrent non-TxForest processes; and (2) it is correct only relative to some assumptions about the file system on which it runs. File system interfaces tend to be informally (under)specified.

We could move TxForest to an existing transactional file system to solve the first issue, though none are in common use. However, we are unaware of any transactional file systems having a formal proof of correctness or even a formal semantics [5, 14, 22, 41]. Therefore, we decided to start from scratch and consider what a new file system that supports transactions could look like. The ideas that we use in TxForest should apply natively, and we pursue that course in this chapter.

We design a new transactional file system Zipper File System (ZFS). This file system is heavily inspired by Kiselyov's work on a file system of the same name [27]. Our goals are to supply provably serializable transactions and a formal semantics. As observed in Chapters 4 and 5, tree zippers, as opposed to the flat maps that are traditionally used, are a natural fit as an underlying abstraction for a file system. Additionally, by designing the file system around zippers, it becomes simple to connect the semantics of the file system to TxForest due to their similarity.

In this chapter, we put the zipper front-and-center, designing a new core language

as the interface to ZFS. Additionally, we provide a formal translation from a core subset of POSIX. In order to provide such a translation, we also formulate a formal semantics for POSIX that mostly matches its informal specification (but with some error cases done away with for simplicity). Incorporating any one of these error cases would be tedious, but not difficult, and could be done in a modular fashion.

The formal semantics of ZFS is split into a local portion, describing the interface for programs written against ZFS, and a global portion, which describes the transaction management piece. We employ optimistic concurrency control to provide serializability among arbitrary ZFS transactions. Any program that interfaces with ZFS is considered a transaction, allowing the system to provide strong guarantees with respect to arbitrary concurrent processes. Additionally, we give a proof of these guarantees.

The translation from POSIX means that ZFS could be run as the root file system, with a POSIX file system as a layer of abstraction above it. This file system could only provide the core POSIX commands specified, but using a second translation layer one could allow a larger set of POSIX commands, most of which can be captured by the core commands (as noted in Gardner *et al*.'s paper on reasoning about POSIX [15]). This would provide a POSIX interface with serializable transactions.

To summarize, the contributions of this chapter are:

1. The design of the Zipper File System, a transactional file system that uses zippers as its underlying abstraction, and a formal semantics describing its behavior.
2. A proof of serializability for this semantics.
3. A semantics for a core subset of POSIX.
4. A translation from this subset of POSIX to ZFS.
5. A prototype implementation of ZFS.

The rest of the chapter proceeds as follows: Section 6.2 formalizes ZFS, describing the local and global semantics. Section 6.3 introduces the POSIX formalization and

| | | |
|---|---|---|
| Strings | $u \in$ | $\Sigma^*$ |
| Names | $a \in$ | *Name* |
| Integers | $n \in$ | $\mathbb{Z}$ |
| Data | $d \in$ | $\mathbb{D}$ |
| Variables | $x \in$ | *Var* |
| Values | $v \in$ | *Val* |
| Environments | $E \in \mathit{Env}$ : | $\mathit{Var} \mapsto \mathit{Val}$ |

**Figure 6.1:** Preliminaries

its translation to ZFS. Section 6.4 describes our prototype implementation of ZFS. Section 6.5 goes through a series of potential extensions that we have considered and Section 6.6 concludes.

## 6.2 The Zipper File System

In this section, we introduce the syntax and file system model of ZFS. We describe the denotational semantics of the local command language before introducing the global transaction manager semantics. Then we state our serializability theorem.

### 6.2.1 Syntax

We present the meta-variables that we use in Figure 6.1 and the syntax of ZFS in Figure 6.2:

- The file system is modeled as a tree. A tree node $t$ is either a directory with a set of named children (Dir $\{\overline{a \times t}\}$) or a file containing some data (File $d$). For a standard implementation, this data would likely be a bitstring, but the formalism admits other typed representations, and it would be quite easy to impose a simple type system on top of the commands given below.
- Paths $p$ are either absolute or relative, denoted by the path's leftmost symbol as `/` and `.` respectively. This symbol is followed by a sequence of path elements $pe$. For

| | | | |
|---|---|---|---|
| Trees | $t$ | ::= | Dir $\{\overline{a \times t}\}$ \| File $d$ |
| Paths | $p$ | ::= | `/` \| `.` \| $p/pe$ |
| Path Elements | $pe$ | ::= | $a$ \| `..` |
| Zippers | $z$ | ::= | {`ancestor` : *Zipper* `option`;<br>`left` : $(a \times t)$ `list`;<br>`focus` : $(a \times t)$;<br>`right` : $(a \times t)$ `list`;} |
| Commands | $c$ | ::= | $zc$ \| `Skip` \| $c_1; c_2$ \| $x := e$ |
| | | \| | `If` $b$ `Then` $c_1$ `Else` $c_2$ \| `While` $b$ `Do` $c$ |
| ZFS Commands | $zc$ | ::= | $zn$ \| $zu$ |
| ZFS Navigations | $zn$ | ::= | `Down` \| `Up` \| `Next` \| `Prev` |
| ZFS Updates | $zu$ | ::= | `Update_File` $e$ \| `RemoveChild` $e$ \| `AddChild` $e_1$ $e_2$ |
| Values | $v$ | ::= | $p$ \| $pe$ \| $a$ \| $d$ \| `true` \| `false` \| ... |
| Expressions | $e, b$ | ::= | $ze$ \| $v$ \| $x$ \| $e_1$ $e_2$ \| ... |
| ZFS Expressions | $ze$ | ::= | $ztf$ \| `Fetch` \| `Name` |
| ZFS Tree Functions | $ztf$ | ::= | `Construct_File` \| `Construct_Dir` \| `Copy` |
| Log Entries | $le$ | ::= | `Write_file` $C_1$ $C_2$ $p$ \| `Read` $C$ $p$ |
| | | \| | `Write_dir` $C_1$ $C_2$ $p$ |
| Logs | $\sigma$ | : | *LogEntry* `list` |

**Figure 6.2:** The syntax of ZFS

a path $p/pe$, the path element $pe$ can either denote a child of the directory at path $p$ called $a$ or the parent directory of $p$ if the path element is the special symbol '`..`'.

- Zippers $z$ are tagged unions representing a focus node and the context surrounding it. The focus node is either the root of the file system or has an `ancestor`; thus the `ancestor` is either a zipper or nothing. The focus node may also have siblings to its `left` and `right`, which are represented as lists of named trees. The first element of both `left` and `right` is the focus node's closest sibling on that side. Finally, the `focus` node itself is a named tree.
- Commands $c$ are the interface to ZFS. They include standard IMP operations as well as the ZFS Commands $zc$ which are divided into two parts:

1. The ZFS Navigations $zn$ which are operations that solely traverse the file system but do not modify the underlying tree. They allow users to move from the focus node `Up` to the parent node, `Down` to the first child, `Next` to the closest sibling on the right, and `Prev` to the closest sibling on the left.
2. The ZFS Updates $zu$ in contrast, are operations that solely modify the tree at the focus node but do not traverse the file system. `Update_File` modifies the file at the focus node to contain its input data. `RemoveChild` takes a name and removes the named child from the directory in focus. `AddChild` takes both a name $a$ and a tree and adds that tree to the focus directory as a child named $a$. Importantly, trees can only be obtained by users with the ZFS Tree Functions mentioned below, which means that `AddChild` does not allow constructing new and arbitrary subtrees in the file system.

- Values $v$ are standard. As are expressions $e$, though they additionally include the ZFS Expressions $ze$. These have three parts:

  1. The ZFS Tree Functions $ztf$ are the only way to produce trees in the language. `Construct_File` and `Construct_Dir` produce an empty file and an empty directory respectively. `Copy` instead returns a copy of the tree at the focus node. Since trees cannot be destructed in the language, this can solely be used with `AddChild` to create a deep copy of the subtree in focus, or, with the addition of `RemoveChild`, to move the subtree.
  2. The `Fetch` expression is reminiscent of `Copy` in that it returns a copy of the focus node's tree. However, it transforms that copy into a more lightweight form, which is destructable, by only including the names of the children of a directory rather than the entire subtree.
  3. The `Name` expression simply returns the name of the focus node.

$$\llbracket \cdot \rrbracket_c^Z : \mathit{Command} \to \mathit{Env} \times \mathit{Zipper} \rightharpoonup (\mathit{Env} \times \mathit{Zipper}) \times \mathit{Log}$$
$$\llbracket \cdot \rrbracket_e^Z : \mathit{Exp} \to \mathit{Env} \times \mathit{Zipper} \to \mathit{Val} \times \mathit{Log}$$

$$\frac{z = (\!|(a', \mathsf{Dir}\ ((a,t) \cdot r))|\!) \qquad z' = z\ \mathit{with}\ (a', \mathsf{Dir}\ \varnothing)}{\llbracket \texttt{Down} \rrbracket_c^Z\ (E, z) = ((E, \wr (a,t) \int^{z'} \rightharpoonup r), \sigma \cdot (\texttt{log\_read}\ z))}$$

$$\frac{z = l \leftharpoonup (\!|f|\!)^{z'} \rightharpoonup r \qquad z' = (\!|(a, \mathsf{Dir}\ \varnothing)|\!)}{\llbracket \texttt{Up} \rrbracket_c^Z\ (E, z) = ((E, z'\ \mathit{with}\ \wr (a, \mathsf{Dir}\ ((\texttt{reverse}\ l) \cdot f \cdot r)) \int), \varepsilon)}$$

$$\frac{z = l \leftharpoonup (\!|f'|\!)^{z'} \rightharpoonup (f \cdot r)}{\llbracket \texttt{Next} \rrbracket_c^Z\ (E, z) = ((E, (f' \cdot l) \leftharpoonup \wr f \int^{z'} \rightharpoonup r), \varepsilon)}$$

$$\frac{z = (f \cdot l) \leftharpoonup (\!|f'|\!)^{z'} \rightharpoonup r}{\llbracket \texttt{Prev} \rrbracket_c^Z\ (E, z) = ((E, l \leftharpoonup \wr f \int^{z'} \rightharpoonup (f' \cdot r)), \varepsilon)}$$

$$\frac{z = (\!|(a, \mathsf{File}\ \_)|\!) \qquad (d, \sigma) = \llbracket e \rrbracket_e^Z\ (E, z)}{\llbracket \texttt{Update\_File} \rrbracket_c^Z\ (E, z) = ((E, z\ \mathit{with}\ (a, \mathsf{File}\ d)), \sigma \cdot (\texttt{log\_file\_write}\ z\ d))}$$

$$\frac{\begin{array}{c}(a', \sigma) = \llbracket e \rrbracket_e^Z\ (E, z) \qquad z = (\!|(a, \mathsf{Dir}\ \ell)|\!) \qquad (a', t) \in \ell \\ \sigma' = \sigma \cdot (\texttt{log\_dir\_RW}\ z\ (\ell \setminus \{(a', t)\}))\end{array}}{\llbracket \texttt{RemoveChild}\ e \rrbracket_c^Z\ (E, z) = ((E, z\ \mathit{with}\ (a, \mathsf{Dir}\ (\ell \setminus \{(a', t)\}))), \sigma')}$$

$$\frac{\begin{array}{c}(a', \sigma) = \llbracket e_1 \rrbracket_e^Z\ (E, z) \qquad (t, \sigma') = \llbracket e_2 \rrbracket_e^Z\ (E, z) \qquad z = (\!|(a, \mathsf{Dir}\ \ell)|\!) \qquad (a', \_) \notin \ell \\ \sigma'' = \sigma \cdot \sigma' \cdot (\texttt{log\_dir\_RW}\ z\ (\ell \cup \{(a', t)\}))\end{array}}{\llbracket \texttt{AddChild}\ e_1\ e_2 \rrbracket_c^Z\ (E, z) = ((E, z\ \mathit{with}\ (a, \mathsf{Dir}\ (\ell \cup \{(a', t)\}))), \sigma'')}$$

**Figure 6.3:** ZFS Command Semantics

- Finally, there are logs $\sigma$, which are lists of log entries. Log entries are records of reads or writes that have occurred, carrying the original value, the modified value (in the case of writes), and the path read or written.

### 6.2.2 Local Semantics

We present the semantics of ZFS in Figures 6.3 and 6.5. Additionally, there are several useful derived commands and expressions provided in Figures 6.4 and 6.6 respectively.

$$
\begin{array}{rcl}
\texttt{GotoChild}\ a & \triangleq & \texttt{Down}; \texttt{While}\ \neg(\texttt{Name} = a)\ \texttt{Do Next} \\
\\
\texttt{Goto /} & \triangleq & \texttt{While}\ \neg\texttt{Is\_Root Do Up} \\
\texttt{Goto .} & \triangleq & \texttt{Skip} \\
\texttt{Goto}\ p/a & \triangleq & \texttt{Goto}\ p; \texttt{GotoChild}\ a \\
\texttt{Goto}\ p/\texttt{..} & \triangleq & \texttt{Goto}\ p; \texttt{Up} \\
\\
\texttt{GoDoReturn /}\ c & \triangleq & x := \texttt{Name}; \texttt{While}\ \neg\texttt{Is\_Root Do}\ (\texttt{Up}; x := \texttt{Name}/x); c; \texttt{Goto}\ x \\
 & & \quad \textbf{where}\ x \text{ is fresh} \\
\texttt{GoDoReturn .}\ c & \triangleq & c \\
\texttt{GoDoReturn}\ p/a\ c & \triangleq & \texttt{GoDoReturn}\ p\ (\texttt{GotoChild}\ a; c; \texttt{Up}) \\
\texttt{GoDoReturn}\ p/\texttt{..}\ c & \triangleq & \texttt{GoDoReturn}\ p\ (x := \texttt{Name}; \texttt{Up}; c; \texttt{GotoChild}\ x) \\
 & & \quad \textbf{where}\ x \text{ is fresh}
\end{array}
$$

**Figure 6.4:** ZFS Derived Commands

At the top of Figure 6.3, the types of the command and expression denotations, $[\![\cdot]\!]^Z_c$ and $[\![\cdot]\!]^Z_e$ respectively, are given. Both take an environment and a zipper pair as input and return a log of their actions on the file system. Commands additionally return a new environment and zipper, while expressions return a value. The inference rules in the rest of the figure show the denotation of different ZFS Commands. Note that the IMP commands are standard and thus excluded.

We repeat Definition 4.2.1 since we will be making heavy use of this zipper notation in this section:

**Definition 4.2.1** (Zipper Notation)**.** We define notation for constructing and deconstructing (*i.e.* pattern matching on) zippers. To construct a zipper we write:

$$\texttt{left} \leftharpoonup \wr \texttt{focus} \int^{z} \rightharpoonup \texttt{right} \triangleq \{\texttt{ancestor} = \texttt{Some}(z); \texttt{left}; \texttt{focus}; \texttt{right}\}$$

where any of `ancestor`, `left`, and `right` can be omitted to denote a zipper with $\texttt{ancestor} = \texttt{None}$, $\texttt{left} = []$, and $\texttt{right} = []$ respectively. For example:

$$\wr \texttt{focus} \int \triangleq \{\texttt{ancestor} = \texttt{None}; \texttt{left} = []; \texttt{focus}; \texttt{right} = []\}$$

Likewise, to destruct a zipper we write:

$$\texttt{left} \leftharpoonup (\!|\texttt{focus}|\!)^{z} \rightharpoonup \texttt{right}$$

where any part can be omitted to ignore that portion of the zipper, but any included part must exist. For example, $z = (\!|\_|\!)^{z'} \iff z.\texttt{ancestor} = \texttt{Some}(z')$.

Finally, to construct a new zipper from an existing zipper and a focus node, we define:

$$z \textit{ with } f \triangleq \{z \texttt{ with focus} = f\}$$

The first four inference rules of Figure 6.3 are the ZFS Navigation Commands $zn$:

- `Down` gets the first child of the focus node and constructs a new zipper focused on it. The new zipper uses a modification of the old zipper as `ancestor`, and the rest of the children of the focus node become `right`. `Down` also returns the log that $e$ produces with a read appended.

  The old zipper gets modified by removing all of its children. This ensures that `ancestors` do not contain stale information: Because the zipper is a functional structure that nonetheless allows local modification, updates have to be propagated upward to `ancestor` nodes. This propagation is done when we step `Up` to an `ancestor`. Unfortunately, with a naïve semantics, this propagation would invalidate various nice properties. For example, if we are at the first child of a directory, we might want to say that if $[\![\texttt{Up}; \texttt{Down}]\!]^{Z}_{c}\ (E, z) = ((E', z'), \_)$ holds then $(E, z) = (E', z')$. However, if a sibling of the current node had changed and the ancestor zipper contained stale information, then the second equality would not hold.

- `Up` makes the `ancestor` the new zipper after properly inserting its updated children into the directory.

- `Next` and `Prev` are symmetric operations. `Next` moves the zipper focus to the right, making the first node of `right` the new `focus` node, and the `focus` node the first left-sibling.

Using these local navigation commands, we can derive three others that move to specific children or paths in the file system, as seen in Figure 6.4:

1. `GotoChild` takes a name as its argument, steps `Down` to the first child of the current focus node, and walks to the `Next` sibling until the focus node has the desired name. `GotoChild` uses the `Name` expression (described below) to find the name of the focus node. If no child of the focus directory has the given name, `GotoChild` will get stuck when it tries to go to the `Next` node and no right-siblings exist.
2. `Goto` takes a path and walks to it with a sequence of `Up` and `GotoChild` commands (along with a while-loop to go to the root if necessary).
3. `GoDoReturn` takes both a path and a command as input and walks to the given path (akin to `Goto`). Once it is at the given path, it executes its input command and walks back from whence it came. `GoDoReturn` uses the name of the focus node to return to it after going `Up`. For example, it reconstructs its current path (in order to return from the root) by interspersing `Up` commands and calls to `Name`. Note that `GoDoReturn` will only work properly if the zipper is in the same location before and after executing the input command.

The three final rules of Figure 6.3 are the ZFS Update Commands $zu$:

- `Update_File` $e$ evaluates $e$ to a data object, which has whatever type you want to support for your files (usually a bitstring). Then, assuming that it is currently at a file, it will change its contents to the result of the evaluation. Simultaneously, `Update_File` logs this write and appends it to the result of its expression evaluation.
- `RemoveChild` $e$ evaluates $e$ to the name of a child of the current directory. It then removes that child from the directory and logs a directory read and write.
- `AddChild` $e_1$ $e_2$ evaluates $e_1$ to the name of a child that is not in the current directory, then it inserts a child with that name whose contents are the result of evaluating

$$\frac{z = (\!|(\_, \mathsf{File}\ d)|\!)}{[\![\texttt{Fetch}]\!]^Z_e\ (E, z) = (File\ d, \texttt{log\_read}\ z)}$$

$$\frac{z = (\!|(\_, \mathsf{Dir}\ \ell)|\!)}{[\![\texttt{Fetch}]\!]^Z_e\ (E, z) = (Dir\ (\texttt{extract\_names}\ \ell), \texttt{log\_read}\ z)}$$

$$\frac{z = (\!|(a, \_)|\!)}{[\![\texttt{Name}]\!]^Z_e\ (E, z) = (a, \texttt{log\_read}\ z)}$$

$$\frac{}{[\![\texttt{Construct\_File}]\!]^Z_e\ (E, z) = (\mathsf{File}\ \varepsilon, \varepsilon)}$$

$$\frac{}{[\![\texttt{Construct\_Dir}]\!]^Z_e\ (E, z) = (\mathsf{Dir}\ \varnothing, \varepsilon)}$$

$$\frac{z = (\!|(\_, t)|\!)}{[\![\texttt{Copy}]\!]^Z_e\ (E, z) = (t, \texttt{log\_read}\ z)}$$

**Figure 6.5:** ZFS Expression Semantics

$e_2$. It also logs a directory read and write.

The ZFS Expressions are shown in Figure 6.5, and they work as follows:

- `Fetch` returns a representation of the focus node. Since all operations are local by design, if the current node is a directory, `Fetch` will only return a list of the names of its children. For a file, it will return its contents. In both cases, a read is logged and the values are properly encapsulated in a sum type.
- `Name` returns the name of the focus node and logs a read.
- `Construct_File` constructs an empty file for use with `AddChild`.
- `Construct_Dir` constructs an empty directory for use with `AddChild`.
- `Copy` is akin to `Fetch`, but instead returns a literal copy of the tree in focus. This tree cannot be destructed, and can thus only be used by `AddChild` to copy or move a whole subtree.

We can derive several useful functions from these expressions as shown in Figure 6.6:

$$
\begin{array}{rcl}
\texttt{Is\_Folder} & \triangleq & (\texttt{Fetch} = \mathit{Dir}\ \_) \\
\texttt{Is\_Root} & \triangleq & (\texttt{Name} = \texttt{/}) \\
\texttt{Has\_Child}\ a & \triangleq & \texttt{match Fetch with} \\
 & & \mid\ \mathit{Dir}\ \ell\ \texttt{when}\ a \in \ell\ \rightarrow \texttt{true} \\
 & & \mid\ \_\ \rightarrow \texttt{false}
\end{array}
$$

**Figure 6.6:** ZFS Derived Expressions

- `Is_Folder` returns a boolean denoting whether the current node is a directory or not.
- `Is_Root` returns a boolean denoting whether the current node is the root or not.
- `Has_Child` takes a name as input and returns a boolean denoting whether the current node has a child with that name.

One important principle behind all of these commands and expressions is that they only perform local queries and modifications of the zipper. They never require any information that is not available at the focus node, nor do they change anything beyond it. This can make each core command and expression a constant time operation.

### 6.2.3 Global Semantics

The global semantics works very similarly to the one in Chapter 5. We first show how to construct a serial, global denotational semantics by lifting the command denotations to bags of transactions, then we construct a concurrent global operational semantics. Finally, we relate the two, proving that ZFS provides serializable transactions.

Figure 6.7 introduces some additional syntax defining what precisely constitutes a transaction. We define the global denotational semantics on a transaction ($[\![]\!]_G$) and a transaction list ($[\![]\!]_{\mathbb{G}}$) respectively as follows:

$$
[\![(((E, \_), c), \_)]\!]_G\ Z \triangleq \texttt{project\_zipper}\ ([\![c]\!]_c^Z\ (E, Z))
$$

$$
[\![\ell]\!]_{\mathbb{G}}\ Z \triangleq \texttt{fold}\ Z\ \ell\ [\![\cdot]\!]_G
$$

| | |
|---|---|
| $ts \in$ *Timestamp* | Timestamps |
| $GL \in$ *TSLog* | Timestamped Logs |
| $td \in$ *Thread* | $\triangleq$ (*Env* $\times$ *Zipper*) $\times$ *Command* |
| $TxS \in$ *TxState* | $\triangleq$ *Command* $\times$ *Timestamp* $\times$ *Log* |
| $t \in$ *Transaction* | $\triangleq$ *Thread* $\times$ *TxState* |
| $T \in$ *Transaction Pool* | $\triangleq$ *Transaction* $Bag$ |

**Figure 6.7:** Global Semantics Additional Syntax

In the first case, we start with a transaction and take a zipper file system as input, producing a new zipper using the denotation of that transaction's command. In the second, we simply lift this operation to act on a list of transactions in the obvious way.

This is a simple way to execute a bag of transactions (with some ordering) on a file system, but it is inherently serial. We want programs to be able to run concurrently on the file system, so this is unacceptable.

Instead, we introduce a global operational semantics that naturally captures all possible interleavings of transactions. In order to produce such a semantics, we should connect it to a local operational semantics that can run individual commands. There is a simple trick that we can use. For the IMP commands, a standard operational semantics will do. For the ZFS commands, we can use the denotational semantics to execute a single step as follows:

$$\frac{((E', z'), \sigma) = [\![zc]\!]^Z_c\ (E, z)}{\langle E, z, zc\rangle \xrightarrow{\sigma}_L \langle E', z', \texttt{Skip}\rangle}$$

This is quite easy to motivate given that all ZFS commands are immediate local modifications and thus inexpensive to consider atomic.

We can now use this local operational semantics to construct a global semantics as seen in Figure 6.8. The global semantics uses three rules to step between two contexts. A context is a tuple of a global file system zipper $Z$, a global log $GL$, and a transaction pool $T$. Intuitively, the global zipper represents the true state of the file system, while each transaction has its own, isolated, local view. The global log records

$$\frac{\langle td\rangle \xrightarrow{\sigma'}_L \langle td'\rangle}{\langle Z, GL, \{(td, (cs, ts, \sigma))\} \uplus T\rangle \to_G \langle Z, GL, \{(td', (cs, ts, \sigma \cdot \sigma'))\} \uplus T\rangle}$$

$$\frac{\texttt{is\_Done?}\ td \qquad \texttt{check\_log}\ GL\ \sigma\ ts \qquad Z' = \texttt{merge}\ Z\ \sigma \qquad GL' = GL \cdot (\texttt{add\_ts}\ \texttt{fresh\_ts}\ \sigma)}{\langle Z, GL, \{(td, (cs, ts, \sigma))\} \uplus T\rangle \to_G \langle Z', GL', T\rangle}$$

$$\frac{\texttt{is\_Done?}\ td \qquad \neg(\texttt{check\_log}\ GL\ \sigma\ ts) \qquad ts' = \texttt{fresh\_ts} \qquad td' = ((\{\}, Z), cs)}{\langle Z, GL, \{(td, (cs, ts, \sigma))\} \uplus T\rangle \to_G \langle Z, GL, \{(td', (cs, ts', []))\} \uplus T\rangle}$$

**Figure 6.8:** Global Operational Semantics

all writes made by committed transactions. The transaction pool contains all currently running transactions. The three rules work as follows:

1. Any transaction in the transaction pool can take a single step in the local semantics, appending the resulting log to its own.
2. If a transaction is finished and does not conflict with previously committed transactions (captured by `check_log`), it can commit. These conflicts occur when a transaction reads stale data. When a transaction commits, it updates the global zipper according to any writes that it has performed (captured by `merge`) and leaves the transaction pool.
3. If a transaction is finished, but conflicts with a previously committed transaction, then it cannot commit and must restart. The transaction gets a fresh timestamp, resets its log and environment, and reruns its original command, $cs$.

Taken together, these rules allow the arbitrary interleaving of transactions, with the ability to roll-back if any issues arise, as is typical of optimistic concurrency control. Using the global denotational and operational semantics, we can now state the serializability theorem. The proof of this theorem is a straightforward modification to the proof Appendix B. The theorem follows:

**Theorem 6.2.1** (Serializability)**.** *Let $Z, Z'$ be file systems, $GL, GL'$ be global logs, and $T$ a transaction pool such that $\forall t \in T.$ `initial` $Z$ $t$, then:*

$$\langle Z, GL, T\rangle \rightarrow_G^* \langle Z', GL', \{\}\rangle \implies \exists \ell \in Perm(T).\ [\![\ell]\!]_{\mathbb{G}}\ Z = Z'$$

*where $\rightarrow_G^*$ is the reflexive, transitive closure of $\rightarrow_G$.*

With this, we have fully characterized the syntax and semantics of ZFS, both within transactions (locally) and between transactions (globally). However, ZFS does not have a standard file system interface and, much like in our previous work, we prefer to meet users where they are rather than require them to rewrite all of their applications. As such, we wish to provide a translation from POSIX down to ZFS, effectively allowing users to get serializable transactions for POSIX.

## 6.3 POSIX Encoding

This section presents a formalization of POSIX and a translation from this formalization into ZFS. Note that this translation has not yet been proven correct.

### 6.3.1 Syntax

We present the syntax of POSIX in Figure 6.9 and use the preliminaries from the previous section (Figure 6.1).

- A file system node $t$ is either a directory containing the names of its children (Dir $\{\overline{a}\}$) or a file containing a string (File $u$).
- Paths and path elements look identical to those of ZFS.
- The file system $fs$ is a partial map from paths to file system nodes. A well-formed file system has three additional properties:

| | | | |
|---|---|---|---|
| File System Nodes | $t$ | ::= | Dir $\{\overline{a}\}$ \| File $u$ |
| Paths | $p$ | ::= | `/` \| `.` \| $p/pe$ |
| Path Elements | $pe$ | ::= | $a$ \| `..` |
| File System | $fs$ | : | *Path* $\rightharpoonup$ *Node* |
| File Descriptors | $fd$ | $\in$ | *FD* |
| Commands | $c$ | ::= | $x := pc$ \| `Skip` \| $c_1; c_2$ \| $x := e$ |
| | | \| | `If` $b$ `Then` $c_1$ `Else` $c_2$ \| `While` $b$ `Do` $c$ |
| POSIX Commands | $pc$ | ::= | `open` $e$ \| `close` $e$ \| `remove` $e$ \| `copy` $e_1$ $e_2$ |
| | | \| | `mkdir` $e$ \| `chdir` $e$ |
| Expressions | $e, b$ | ::= | $pf$ \| $v$ \| $x$ \| $e_1$ $e_2$ \| ... |
| Values | $v$ | ::= | $fd$ \| $p$ \| $s$ \| `true` \| `false` \| ... |
| POSIX Functions | $pf$ | ::= | `write` $e_1$ $e_2$ \| `read` $e_1$ $e_2$ \| `seek` $e_1$ $e_2$ |
| POSIX Statuses | $s$ | ::= | **`SUCCESS`** \| **`EBADF`** \| **`EISDIR`** \| **`ENOTDIR`** |
| | | \| | **`ENOTEMPTY`** \| **`EEXIST`** \| **`ENOENT`** |

**Figure 6.9:** POSIX Syntax

**Definition 6.3.1** (Well-Formedness)**.** A file system $fs$ is well-formed if and only if:

1. $fs(/) = \text{Dir } \_$
2. $p/a \in fs \iff fs(p) = \text{Dir } \ell \ \wedge \ a \in \ell$
3. $p \in fs \implies \neg(\texttt{..} \in p \vee \texttt{.} \in p)$

The first two restrictions assert that file systems are trees rather than arbitrary partial maps. The last excludes relative paths and those containing the special parent symbol (`..`) from being mapped. For the remainder of this chapter, we only consider well-formed file systems.

- File descriptors $fd$ are effectively abstract unique identifiers. For example, in a typical POSIX implementation, file descriptors are simply integers. In this formalization, they identify (*i.e.* point to in the file descriptor map) a path, its representation, and the offset from which reads and writes will proceed.

- Commands $c$ are made up of the standard IMP commands and $x := pc$, which runs a POSIX Command $pc$ and stores its result in a variable $x$. The POSIX commands are a subset of those in the POSIX standard [24]:

    1. The `open` command takes a path and creates a file descriptor from it, possibly creating a file if none exist.
    2. The `close` command takes an open file descriptor and closes it, potentially updating the file system based on changes made.
    3. The `remove` command takes the path of an empty directory or a file and removes it from the file system.
    4. The `copy` command takes the path of a subtree to be copied and an unmapped path to which the subtree is copied.
    5. The `mkdir` command takes an unmapped path and creates a directory there.
    6. The `chdir` command takes a path to a directory and makes that the new working path.

- Expressions $e$ and Values $v$ are standard, with the exception of the POSIX functions. These are `write`, `read`, and `seek`:

    1. The `write` function takes an open file descriptor and a string as its arguments and writes the string to the offset and file specified in the file descriptor.
    2. The `read` function takes an open file descriptor and an integer $n$ and reads $n$ characters/entries from the file/directory in the file descriptor at the specified offset.
    3. The `seek` function takes an open file descriptor and an integer $n$ and sets the file descriptors offset to $n$.

- The POSIX Statuses $s$ are a subset of those in the POSIX standard. They indicate the type of error encountered or, in the case of **`SUCCESS`**, success. In standard POSIX implementations, statuses are represented by integers.

With that, we can move on to the formal semantics of the commands and functions.

### 6.3.2 Semantics

This subsection introduces the semantics of POSIX Commands and Functions. We elide the IMP commands, since they are standard. The semantics is split into successful operation and errors. We present each in turn.

Figure 6.10 contains the denotational semantics of the POSIX commands. The denotation of a POSIX command in a context is a pair of a value and a new context. The $x := pc$ command thus runs a POSIX command $pc$, assigns the returned value to $x$, and updates the state of the world to the returned context.

A context consists of an environment $E$, the current working path $wp$, a map of open file descriptors *FD*, and a file system $fs$. The file descriptor map *FD* maps open file descriptors to a tuple of the path that they point to, a buffered value, and an offset. The buffered value is read and written at the offset until finally being persisted back to the file system when the file descriptor is closed.

The denotation of expressions $[\![\cdot]\!]^{\mathbb{P}}_{e}$, also takes a full context as input, but outputs a pair of the expression's return value and a possibly updated file descriptor map. The file descriptor map is the only part of a context that an expression can modify.
The six POSIX commands in Figure 6.10 work as follows:

- The `open` command evaluates its argument to a path, which it then canonicalizes by combining it with the current working path. If the canonical path is mapped in the file system, then `open` generates a fresh file descriptor and maps the path, its contents, and the starting offset ($0$) to it.

$$\frac{(p', \mathit{FD}_1) = [\![e]\!]^{\mathbb{P}}_{e}\ (E, wp, \mathit{FD}, fs) \qquad p = \texttt{canonicalize}\ wp\ p' \qquad p \in fs \qquad fd = \texttt{fresh}\ \mathit{FD}_1 \qquad \mathit{FD}_2 = \mathit{FD}_1[fd \mapsto (p, fs(p), 0)]}{[\![\texttt{open}\ e]\!]^{\mathbb{P}}_{pc}\ (E, wp, \mathit{FD}, fs) = (fd, (E, wp, \mathit{FD}_2, fs))}$$

$$\frac{\begin{array}{c}(p', \mathit{FD}_1) = [\![e]\!]^{\mathbb{P}}_{e}\ (E, wp, \mathit{FD}, fs) \qquad p/a = \texttt{canonicalize}\ wp\ p' \qquad fs(p) = \mathsf{Dir}\ \ell \\ a \notin \ell \qquad fd = \texttt{fresh}\ \mathit{FD}_1 \qquad \mathit{FD}_2 = \mathit{FD}_1[fd \mapsto (p/a, \mathsf{File}\ \varepsilon, 0)] \\ fs_1 = fs[p \mapsto \mathsf{Dir}\ \ell \cup \{a\}] \qquad fs_2 = fs_1[p/a \mapsto \mathsf{File}\ \varepsilon]\end{array}}{[\![\texttt{open}\ e]\!]^{\mathbb{P}}_{pc}\ (E, wp, \mathit{FD}, fs) = (fd, (E, wp, \mathit{FD}_2, fs_2))}$$

$$\frac{(fd, \mathit{FD}_1) = [\![e]\!]^{\mathbb{P}}_{e}\ (E, wp, \mathit{FD}, fs) \qquad \mathit{FD}_1(fd) = (p, \mathsf{File}\ u, \_) \qquad fs(p) = \mathsf{File}\ \_ \qquad \mathit{FD}_2 = \mathit{FD}_1[fd \mapsto \bot] \qquad fs_1 = fs[p \mapsto \mathsf{File}\ u]}{[\![\texttt{close}\ e]\!]^{\mathbb{P}}_{pc}\ (E, wp, \mathit{FD}, fs) = (\textbf{SUCCESS}, (E, wp, \mathit{FD}_2, fs_1))}$$

$$\frac{(fd, \mathit{FD}_1) = [\![e]\!]^{\mathbb{P}}_{e}\ (E, wp, \mathit{FD}, fs) \qquad \mathit{FD}_1(fd) = (p, t, \_) \qquad (t = \mathsf{Dir}\ \_) \vee (fs(p) \neq \mathsf{File}\ \_) \qquad \mathit{FD}_2 = \mathit{FD}_1[fd \mapsto \bot]}{[\![\texttt{close}\ e]\!]^{\mathbb{P}}_{pc}\ (E, wp, \mathit{FD}, fs) = (\textbf{SUCCESS}, (E, wp, \mathit{FD}_2, fs))}$$

$$\frac{\begin{array}{c}(p', \mathit{FD}_1) = [\![e]\!]^{\mathbb{P}}_{e}\ (E, wp, \mathit{FD}, fs) \qquad p/a = \texttt{canonicalize}\ wp\ p' \\ fs(p) = \mathsf{Dir}\ \ell \qquad (fs(p/a) = \mathsf{Dir}\ \varnothing) \vee (fs(p/a) = \mathsf{File}\ \_) \\ fs_1 = fs[p \mapsto \mathsf{Dir}\ \ell \setminus \{a\}] \qquad fs_2 = fs_1[p/a \mapsto \bot]\end{array}}{[\![\texttt{remove}\ e]\!]^{\mathbb{P}}_{pc}\ (E, wp, \mathit{FD}, fs) = (\textbf{SUCCESS}, (E, wp, \mathit{FD}_1, fs_2))}$$

$$\frac{\begin{array}{c}(p_1, \mathit{FD}_1) = [\![e_1]\!]^{\mathbb{P}}_{e}\ (E, wp, \mathit{FD}, fs) \qquad (p_2, \mathit{FD}_2) = [\![e_2]\!]^{\mathbb{P}}_{e}\ (E, wp, \mathit{FD}_1, fs) \\ p' = \texttt{canonicalize}\ wp\ p_1 \qquad p' \in fs \\ p/a = \texttt{canonicalize}\ wp\ p_2 \qquad fs(p) = \mathsf{Dir}\ \ell \qquad a \notin \ell \\ fs_1 = fs[p \mapsto \mathsf{Dir}\ \ell \cup \{a\}] \qquad fs_2 = fs_1[p/a \mapsto fs(p')]\end{array}}{[\![\texttt{copy}\ e_1\ e_2]\!]^{\mathbb{P}}_{pc}\ (E, wp, \mathit{FD}, fs) = (\textbf{SUCCESS}, (E, wp, \mathit{FD}_2, fs_2))}$$

$$\frac{\begin{array}{c}(p', \mathit{FD}_1) = [\![e]\!]^{\mathbb{P}}_{e}\ (E, wp, \mathit{FD}, fs) \qquad p/a = \texttt{canonicalize}\ wp\ p' \qquad fs(p) = \mathsf{Dir}\ \ell \\ a \notin \ell \qquad fs_1 = fs[p \mapsto \mathsf{Dir}\ \ell \cup \{a\}] \qquad fs_2 = fs_1[p/a \mapsto \mathsf{Dir}\ \varnothing]\end{array}}{[\![\texttt{mkdir}\ e]\!]^{\mathbb{P}}_{pc}\ (E, wp, \mathit{FD}, fs) = (\textbf{SUCCESS}, (E, wp, \mathit{FD}_1, fs_2))}$$

$$\frac{(p', \mathit{FD}_1) = [\![e]\!]^{\mathbb{P}}_{e}\ (E, wp, \mathit{FD}, fs) \qquad wp' = \texttt{canonicalize}\ wp\ p' \qquad fs(wp') = \mathsf{Dir}\ \_}{[\![\texttt{chdir}\ e]\!]^{\mathbb{P}}_{pc}\ (E, wp, \mathit{FD}, fs) = (\textbf{SUCCESS}, (E, wp', \mathit{FD}_1, fs))}$$

**Figure 6.10:** POSIX Command Semantics

If the canonical path is not mapped, but its parent is a directory, then open constructs a new file at that path, updating the file system appropriately, and generates a file descriptor containing an empty file.

Finally, `open` returns the generated file descriptor and a context with an updated file descriptor map and possibly an updated file system.

- The `close` command evaluates its argument to an open file descriptor, from which it gets a path and buffered tree. If the tree is a file and the path is mapped to a file in the file system, then the path in the file system is updated to contain the buffered value. If, instead, the path is unmapped in the file system or the tree is a directory, then the file system remains unchanged. In either situation, the file descriptor is unmapped and the new context is returned with a **`SUCCESS`** status.
- The `remove` command evaluates its argument to a path, which it then canonicalizes by combining it with the current working path. It ensures that the canonical path maps to a file or an empty directory in the file system, and removes the mapping from the file system, while keeping it well-formed.
- The `copy` command evaluates its arguments to two paths, which it then canonicalizes. It ensures that the first path is mapped in the file system, that the second path is not, and that the second path's parent is a directory. The command then maps the second path to the contents of the first, properly adding a new child to the parent directory. Note that the combination of `copy` and `remove` can be used to implement `rename`.
- The `mkdir` command evaluates its argument to a path, which it then canonicalizes by combining it with the current working path. It ensures that the canonical path is unmapped while its parent is a directory. Then it maps that path to an empty directory while keeping the file system well-formed.

$$\frac{(fd, \mathit{FD}_1) = [\![e_1]\!]^{\mathbb{P}}_e\ (E, wp, \mathit{FD}, fs) \qquad (u', \mathit{FD}_2) = [\![e_2]\!]^{\mathbb{P}}_e\ (E, wp, \mathit{FD}_1, fs) \\ \mathit{FD}_2(fd) = (p, \mathsf{File}\ u, n) \qquad u'' = \texttt{substitute\_at}\ u\ n\ u' \\ n' = n + \texttt{length}\ u' \qquad \mathit{FD}_3 = \mathit{FD}_2[fd \mapsto (p, \mathsf{File}\ u'', n')]}{[\![\texttt{write}\ e_1\ e_2]\!]^{\mathbb{P}}_e\ (E, wp, \mathit{FD}, fs) = (\textbf{SUCCESS}, \mathit{FD}_3)}$$

$$\frac{(fd, \mathit{FD}_1) = [\![e_1]\!]^{\mathbb{P}}_e\ (E, wp, \mathit{FD}, fs) \qquad (n', \mathit{FD}_2) = [\![e_2]\!]^{\mathbb{P}}_e\ (E, wp, \mathit{FD}_1, fs) \\ \mathit{FD}_2(fd) = (p, \mathsf{File}\ u, n) \\ n'' = \texttt{min}\ (\texttt{length}\ u)\ (n + n') \qquad \mathit{FD}_3 = \mathit{FD}_2[fd \mapsto (p, \mathsf{File}\ u, n'')]}{[\![\texttt{read}\ e_1\ e_2]\!]^{\mathbb{P}}_e\ (E, wp, \mathit{FD}, fs) = (u[n : n''], \mathit{FD}_3)}$$

$$\frac{(fd, \mathit{FD}_1) = [\![e_1]\!]^{\mathbb{P}}_e\ (E, wp, \mathit{FD}, fs) \qquad (n', \mathit{FD}_2) = [\![e_2]\!]^{\mathbb{P}}_e\ (E, wp, \mathit{FD}_1, fs) \\ \mathit{FD}_2(fd) = (p, \mathsf{Dir}\ \ell, n) \\ n'' = \texttt{min}\ (\texttt{length}\ \ell)\ (n + n') \qquad \mathit{FD}_3 = \mathit{FD}_2[fd \mapsto (p, \mathsf{Dir}\ \ell, n'')]}{[\![\texttt{read}\ e_1\ e_2]\!]^{\mathbb{P}}_e\ (E, wp, \mathit{FD}, fs) = (\ell[n : n''], \mathit{FD}_3)}$$

$$\frac{(fd, \mathit{FD}_1) = [\![e_1]\!]^{\mathbb{P}}_e\ (E, wp, \mathit{FD}, fs) \qquad (n, \mathit{FD}_2) = [\![e_2]\!]^{\mathbb{P}}_e\ (E, wp, \mathit{FD}_1, fs) \\ \mathit{FD}_2(fd) = (p, t, \_) \qquad \mathit{FD}_3 = \mathit{FD}_2[fd \mapsto (p, t, n)]}{[\![\texttt{seek}\ e_1\ e_2]\!]^{\mathbb{P}}_e\ (E, wp, \mathit{FD}, fs) = (\textbf{SUCCESS}, \mathit{FD}_3)}$$

**Figure 6.11:** POSIX Function Semantics

- The `chdir` command evaluates its argument to a path, which it then canonicalizes by combining it with the current working path. It ensures that the canonical path maps to a directory in the file system, and updates the working path to point to it.

Figure 6.11 defines the semantics of the POSIX functions, `write`, `read`, and `seek`. They work as follows:

- The `write` function evaluates its arguments to an open file descriptor, which contains a file, and a string. It then writes the input string to the string contained in the file, starting at offset $n$, using the `substitute_at` function. It appropriately updates the open file descriptor and returns **SUCCESS**.
- The `read` function evaluates its arguments to an open file descriptor and an integer. The input integer $n'$ determines the max number of characters or directory entries to be read. Starting from the offset in the file descriptor, at most $n'$ characters or entries are returned and the file descriptor is modified accordingly.

$$\frac{X \in \{\texttt{open}\ e,\ \texttt{remove}\ e,\ \texttt{copy}\ e\ \_,\ \texttt{copy}\ \_\ e,\ \texttt{mkdir}\ e,\ \texttt{chdir}\ e\} \quad (p', FD_1) = [\![e]\!]^{\mathbb{P}}_e\ (E, wp, FD, fs) \quad p/\ldots/pe = \texttt{canonicalize}\ wp\ p' \quad p \notin fs}{[\![X]\!]^{\mathbb{P}}_{pc}\ (E, wp, FD, fs) = (\textbf{ENOENT}, (E, wp, FD_1, fs))}$$

$$\frac{X \in \{\texttt{open}\ e,\ \texttt{remove}\ e,\ \texttt{copy}\ e\ \_,\ \texttt{copy}\ \_\ e,\ \texttt{mkdir}\ e,\ \texttt{chdir}\ e\} \quad (p', FD_1) = [\![e]\!]^{\mathbb{P}}_e\ (E, wp, FD, fs) \quad p/\ldots/pe = \texttt{canonicalize}\ wp\ p' \quad fs(p) = \mathsf{File}\ u}{[\![X]\!]^{\mathbb{P}}_{pc}\ (E, wp, FD, fs) = (\textbf{ENOTDIR}, (E, wp, FD_1, fs))}$$

$$\frac{X \in \{\texttt{remove}\ e,\ \texttt{copy}\ e\ \_,\ \texttt{chdir}\ e\} \quad (p', FD_1) = [\![e]\!]^{\mathbb{P}}_e\ (E, wp, FD, fs) \quad p = \texttt{canonicalize}\ wp\ p' \quad p \notin fs}{[\![X]\!]^{\mathbb{P}}_{pc}\ (E, wp, FD, fs) = (\textbf{ENOENT}, (E, wp, FD_1, fs))}$$

$$\frac{(fd, FD_1) = [\![e]\!]^{\mathbb{P}}_e\ (E, wp, FD, fs) \quad fd \notin FD_1}{[\![\texttt{close}\ e]\!]^{\mathbb{P}}_{pc}\ (E, wp, FD, fs) = (\textbf{EBADF}, (E, wp, FD_1, fs))}$$

$$\frac{(p', FD_1) = [\![e]\!]^{\mathbb{P}}_e\ (E, wp, FD, fs) \quad p = \texttt{canonicalize}\ wp\ p' \quad fs(p) = \mathsf{Dir}\ \ell\ \wedge\ \ell \neq \varnothing}{[\![\texttt{remove}\ e]\!]^{\mathbb{P}}_{pc}\ (E, wp, FD, fs) = (\textbf{ENOTEMPTY}, (E, wp, FD_1, fs))}$$

$$\frac{X \in \{\texttt{copy}\ \_\ e,\ \texttt{mkdir}\ e\} \quad (p', FD_1) = [\![e]\!]^{\mathbb{P}}_e\ (E, wp, FD, fs) \quad p = \texttt{canonicalize}\ wp\ p' \quad p \in fs}{[\![X]\!]^{\mathbb{P}}_{pc}\ (E, wp, FD, fs) = (\textbf{EEXIST}, (E, wp, FD_1, fs))}$$

$$\frac{(p', FD_1) = [\![e]\!]^{\mathbb{P}}_e\ (E, wp, FD, fs) \quad p = \texttt{canonicalize}\ wp\ p' \quad fs(p) = \mathsf{File}\ u}{[\![\texttt{chdir}\ e]\!]^{\mathbb{P}}_{pc}\ (E, wp, FD, fs) = (\textbf{ENOTDIR}, (E, wp, FD_1, fs))}$$

---

$$\frac{X \in \{\texttt{write}\ e_1\ e_2,\ \texttt{read}\ e_1\ e_2,\ \texttt{seek}\ e_1\ e_2\} \quad (fd, FD_1) = [\![e_1]\!]^{\mathbb{P}}_e\ (E, wp, FD, fs) \quad fd \notin FD_1}{[\![X]\!]^{\mathbb{P}}_e\ (E, wp, FD, fs) = (\textbf{EBADF}, FD_1)}$$

$$\frac{(fd, FD_1) = [\![e_1]\!]^{\mathbb{P}}_e\ (E, wp, FD, fs) \quad FD_1(fd) = (p, \mathsf{Dir}\ \ell, \_)}{[\![\texttt{write}\ e_1\ e_2]\!]^{\mathbb{P}}_e\ (E, wp, FD, fs) = (\textbf{EISDIR}, FD_1)}$$

**Figure 6.12:** POSIX Command and Function Error Semantics

- The `seek` function evaluates its arguments to an open file descriptor and an integer. The input integer is set as the new offset for the file descriptor.

Figure 6.12 defines the conditions in which commands and functions cause POSIX errors:

- All commands except `close` share the need for a valid path. If some strict prefix

of their input path is unmapped in the file system or maps to a file, then they will return **ENOENT** or **ENOTDIR** respectively.

- The `remove`, `copy`, and `chdir` commands additionally require that their input path (rather than just all strict prefixes) must be mapped in the file system. Otherwise they return **ENOENT**. For `copy`, this is only true of its first argument.
- The `close` command is the only one that takes a file descriptor as input. If that file descriptor is not open (*i.e.* mapped in the file descriptor map), then it returns **EBADF**.
- The `remove` command has additional constraints on its mapped input path. In particular, it cannot remove non-empty directories. If the path is a non-empty directory, `remove` returns **ENOTEMPTY**.
- The `copy` and `mkdir` command both require an input path that is not mapped in the file system. If it is mapped, they return **EEXIST**. For `copy`, this is only true of its second argument.
- The `chdir` command accepts only paths that point to directories. If it gets a path pointing to a file, then instead it returns **ENOTDIR**.
- All functions require their first argument to be an open file descriptor. If it is not, then they return **EBADF**.
- The `write` function further requires that the open file descriptor does not point to a directory. Otherwise, it returns **EISDIR**.

Unlike the success rules, the failure rules are non-deterministic. So multiple error conditions could hold at the same time. As such, a natural question is whether there should be a precedence order for delivery. In the POSIX specification, no such ordering exists and thus I have not defined one here, but it would be straightforward to choose a precedence or make the rules non-overlapping. Every implementation I have seen

evaluates paths one entry at a time, and the first error reported would thus simply be the first error condition encountered.

That concludes the subsection on POSIX semantics. We can now define a translation from the POSIX semantics to the ZFS semantics, allowing us to run POSIX on top of ZFS to get serializability guarantees.

### 6.3.3 Translation Semantics

This subsection defines a translation from POSIX to ZFS. This translation lets programs use a (subset of a) standard POSIX API, while gaining the benefits of ZFS's concurrency guarantees.

Figure 6.13 defines how expressions are translated from POSIX to ZFS. An expression in POSIX is translated to a pair of a ZFS command and a ZFS expression. Intuitively, because expressions in POSIX can change the state of the file descriptor map, which we represent as a special variable in ZFS, they must at least use assignments in order to be properly translated.

In the translation, the file descriptor map is denoted $\$\texttt{posix\_fds}$. All variables starting with $\$$ are fresh (except for $\$\texttt{posix\_fds}$ and $\$RET$) and not available to users. $\$RET$ is a designated variable that captures the return values of commands. The return value of a translated POSIX expression is captured by the ZFS expression. In the figures, meta-variables are typeset in boldface. For example, **fd** is a meta-file descriptor, which represents an actual file descriptor variable. No meta-variables remain after a full translation.

In Figure 6.13, the application expression gives a flavor of how these translations would look for standard expressions. The POSIX functions are translated using an auxiliary function (**check_fd_do**) defined in Figure 6.16. The auxiliary function translates and evaluates the first expression, checks that the resulting file descriptor is open, and

$$\boxed{\begin{array}{l}(\!|\cdot|\!)_e : \mathit{PExp} \to (\mathit{ZFSCommand} \times \mathit{ZFSExp}) \\ (\!|\cdot|\!)_{pf} : \mathit{PFun} \to (\mathit{ZFSCommand} \times \mathit{ZFSExp})\end{array}}$$

$$\begin{array}{l}
(\!|e_1\ e_2|\!)_e \triangleq \\
\quad (\mathbf{c_3}, \mathbf{e_3}) \texttt{ = } (\!|e_2|\!)_e \\
\quad (\mathbf{c_4}, \mathbf{e_4}) \texttt{ = } (\!|e_1|\!)_e \\
\quad (\mathbf{c_3}; \$x := \mathbf{e_3}; \mathbf{c_4}, \mathbf{e_4}\ \$x)
\end{array}$$

$$\begin{array}{lcl}
(\!|\texttt{write}\ e_1\ e_2|\!)_{pf} & \triangleq & \texttt{(}\mathbf{check_fd_do}\ e_1\ \texttt{(}\mathbf{write}\ e_2\texttt{),}\$RET\texttt{)} \\
(\!|\texttt{read}\ e_1\ e_2|\!)_{pf} & \triangleq & \texttt{(}\mathbf{check_fd_do}\ e_1\ \texttt{(}\mathbf{read}\ e_2\texttt{) ,}\$RET\texttt{)} \\
(\!|\texttt{seek}\ e_1\ e_2|\!)_{pf} & \triangleq & \texttt{(}\mathbf{check_fd_do}\ e_1\ \texttt{(}\mathbf{seek}\ e_2\texttt{) ,}\$RET\texttt{)}
\end{array}$$

$$\begin{array}{l}
\mathbf{write\ e\ fd} \triangleq \\
\quad (\mathbf{c_1}, \mathbf{e_1}) \texttt{ = } (\!|\mathbf{e}|\!)_e \\
\quad (\$p, \$t, \$n) := \$\texttt{posix_fds}(\mathbf{fd}); \\
\quad \texttt{If}\ \ \$t = \mathit{Dir}\ _ \\
\quad \texttt{Then}\ \ \$RET := \mathbf{EISDIR} \\
\quad \texttt{Else}\ \ \mathbf{c_1}; \$u' := \mathbf{e_1}; \\
\qquad\quad \mathit{File}\ \$u := \$t; \\
\qquad\quad \$u'' := \texttt{substitute_at}\ \$u\ \$n\ \$u'; \\
\qquad\quad \$n' := \$n + \texttt{length}\ \$u'; \\
\qquad\quad \$\texttt{posix_fds}(\mathbf{fd}) := (\$p, \mathit{File}\ \$u'', \$n') \\
\qquad\quad \$RET := \mathbf{SUCCESS}
\end{array}$$

$$\begin{array}{l}
\mathbf{read\ e\ fd} \triangleq \\
\quad (\mathbf{c_1}, \mathbf{e_1}) \texttt{ = } (\!|\mathbf{e}|\!)_e \\
\quad (\$p, \$t, \$n) := \$\texttt{posix_fds}(\mathbf{fd}); \\
\quad \mathbf{c_1}; \$n' := \mathbf{e_1}; \\
\quad (\$contents, \$RET) := \texttt{match}\ \$t\ \texttt{with} \\
\quad |\ \ \mathit{File}\ \$u\ \texttt{when}\ \$n'' = \texttt{min}\ (\texttt{length}\ \$u)\ (\$n + \$n')\ \rightarrow\ ((\$p, \$t, \$n''), \$u[\$n : \$n'']) \\
\quad |\ \ \mathit{Dir}\ \$\ell\ \texttt{when}\ \$n'' = \texttt{min}\ (\texttt{length}\ \$\ell)\ (\$n + \$n')\ \rightarrow\ ((\$p, \$t, \$n''), \$\ell[\$n : \$n'']) \\
\quad |\ \ _\ \rightarrow\ (\$\texttt{posix_fds}(\mathbf{fd}), \mathbf{ENOENT}) \\
\quad \$\texttt{posix_fds}(\mathbf{fd}) := \$contents
\end{array}$$

$$\begin{array}{l}
\mathbf{seek\ e\ fd} \triangleq \\
\quad (\mathbf{c_1}, \mathbf{e_1}) \texttt{ = } (\!|\mathbf{e}|\!)_e \\
\quad (\$p, \$t, _) := \$\texttt{posix_fds}(\mathbf{fd}); \\
\quad \mathbf{c_1}; \$n := \mathbf{e_1}; \\
\quad \$\texttt{posix_fds}(\mathbf{fd}) := (\$p, \$t, \$n); \\
\quad \$RET := \mathbf{SUCCESS}
\end{array}$$

**Figure 6.13:** POSIX Function Translation into ZFS. All variables starting with $ are unavailable for use by users. They are also all fresh except for $posix_fds, which is a designated variable for the File Descriptor map. $RET is a designated variable for the return value of the function.

$(\!|\cdot|\!)_{pc} : PCommand \to Command$

$(\!|x := pc|\!)_c \triangleq (\!|pc|\!)_{pc}; x := \$RET$

$(\!|\texttt{close}\ e|\!)_{pc} \triangleq$ **check_fd_do** $e$ **close**

**close_local a u** $\triangleq$
  If Has_Child **a**
  Then GotoChild **a**;
       If $\neg$Is_Folder
       Then Update_File **u**
       Else Skip
  Else Skip

**close fd** $\triangleq$
  $(\$p/\$a, \$t, \$n) := \$\texttt{posix\_fds}(\textbf{fd})$;
  $\$\texttt{posix\_fds}(\textbf{fd}) := \bot$;
  If $\$t = File\ \$u$
  Then **check_do_return** $\$p$ (**close_local** $\$a\ \$u$)
  Else Skip;
  $\$RET :=$ **SUCCESS**

$(\!|\texttt{open}\ e|\!)_{pc} \triangleq$
  $(\textbf{c}_1, \textbf{e}_1)$ = $(\!|e|\!)_e$
  $\textbf{c}_1; \$p/\$a :=$ canonicalize $\textbf{e}_1$;
  **check_do_return** $\$p$ (**open** $\$p/\$a$)

**open p/a** $\triangleq$
  If Is_Folder
  Then If Has_Child **a**
       Then GotoChild **a**;
            $\$t :=$ Fetch
            Up
       Else AddChild **a** Construct_File;
            $\$t := File\ \varepsilon$
       $\$fd :=$ fresh $\$\texttt{posix\_fds}$;
       $\$\texttt{posix\_fds}(\$fd) := (\textbf{p/a}, \$t, 0)$
       $\$RET := \$fd$
  Else $\$RET :=$ **ENOTDIR**

**Figure 6.14:** POSIX Command Translation into ZFS (1). All variables starting with $ are unavailable for use by users. $\$RET$ is a designated variable for the return value of the function.

passes it to the meta-function that is its second argument.

Each of **write**, **read**, and **seek**, given this file descriptor and the second argument of their namesakes as input, combine the success and error semantics of POSIX given above and execute it in ZFS.

Figures 6.14 and 6.15 capture the translations of POSIX commands, as well as the sole interesting non-POSIX command. The IMP commands have the standard translation.

Many commands use **check_do_return** as a sub-routine, which effectively captures the first two rules of Figure 6.12 and gets the ZFS zipper to where it needs to make changes. The command passed in as the second argument of **check_do_return** actually performs most of the details of open, close, remove, copy, and mkdir. After executing its

```
⦇mkdir e⦈_pc ≜
  (c1, e1) = ⦇e⦈_e
  c1; $p/$a := canonicalize e1;
  $t := Construct_Dir
  check_do_return $p (mknode $a $t)

mknode a t ≜
  If Is_Folder
  Then If Has_Child a
       Then $RET := EEXIST
       Else AddChild a t;
            $RET := SUCCESS
  Else $RET := ENOTDIR

⦇copy e1 e2⦈_pc ≜
  (c3, e3) = ⦇e1⦈_e
  (c4, e4) = ⦇e2⦈_e
  c3; $p' := canonicalize e3;
  $RET := SUCCESS;
  check_do_return $p' ($t := Copy);
  If $RET = SUCCESS
  Then c4; $p/$a := canonicalize e4;
       check_do_return $p (mknode $a $t);
  Else Skip
```

```
⦇remove e⦈_pc ≜
  (c1, e1) = ⦇e⦈_e
  c1; $p/$a := canonicalize e1;
  check_do_return $p (remove $a)

remove a ≜
  $RET := SUCCESS;
  cdr_aux ($x := Fetch) [a]
  If $RET = SUCCESS
  Then If ($x = Dir ℓ) ∧ (ℓ ≠ ∅)
       Then $RET := ENOTEMPTY
       Else RemoveChild a
  Else Skip

⦇chdir e⦈_pc ≜
  (c1, e1) = ⦇e⦈_e
  c1; $p := canonicalize e1;
  $RET := SUCCESS;
  check_do_return $p ($b := Is_Folder);
  If $RET = SUCCESS
  Then If $b
       Then Goto $p
       Else $RET := ENOTDIR
  Else Skip
```

**Figure 6.15:** POSIX Command Translation into ZFS (2). All variables starting with $ are unavailable for use by users. $\$RET$ is a designated variable for the return value of the function.

input command, **check_do_return** returns back from whence it came, thus leaving the working path of POSIX, which corresponds to the zipper's position in the tree, unchanged. For chdir, **check_do_return** is entirely used to check for errors, and it instead changes the working path after **check_do_return** has returned.

This concludes both the subsection on the translation from POSIX to ZFS, and the entire section on POSIX. Next, we will briefly describe the current implementation of ZFS in OCaml.

```
cdr_aux c [] ≜ c
cdr_aux c (a · l) ≜
  If Is_Folder
  Then If Has_Child a
       Then GoDoReturn a (cdr_aux c l)
       Else $RET := ENOENT
  Else $RET := ENOTDIR

check_do_return p c ≜
  hd · l = listify_path p
  GoDoReturn hd (cdr_aux c l)
```

```
check_fd_do e f ≜
  (c1, e1) = (|e|)_e
  c1; $fd := e1;
  If $fd ∉ $posix_fds
  Then $RET := EBADF
  Else f $fd
```

**Figure 6.16:** Helper functions for POSIX → ZFS Translation

## 6.4 Implementation

We have built a prototype implementation of ZFS in 1047 lines of OCaml. It implements most of the functionality described in Section 6.2.

The current implementation uses an underlying POSIX file system and is available only as an application, so all users would be required to run their programs through it manually, which is tedious at best. However, a minor engineering effort would suffice to set it up as an API; a moderate engineering effort would be required to implement a FUSE version.

One of the most interesting features of the implementation is that the zipper is functional, which allows us to employ copy-on-write semantics. Thus, moving or copying a whole subtree using some combination of `Copy`, `RemoveChild`, and `AddChild` is effectively instantaneous. Additionally, a copied subtree need not take any extra space until it is changed in some way, and even then, only the tree changes need to be stored.

## 6.5 Future Work

This section introduces some extensions to further improve ZFS, usually in terms of theoretical improvements with practical consequences.

There are three near-term extensions that we have considered for the POSIX encoding:

1. While we believe that the POSIX to ZFS translation that is presented in Section 6.3.3 is correct, we have not yet proven this fact. The goal would be to prove a bisimulation relation.

2. We have presented a POSIX to ZFS translation, but it would also be useful to have a ZFS to POSIX translation. Additionally, this should be significantly easier than the direction we have gone. With such a translation, it would become possible to run ZFS on top of a normal POSIX file system. This is particularly attractive since one of the goals of this entire thesis is to meet programmers where they are. PADS, Forest, iForest, and TxForest all work with data that is already there, while trying to make it easier for users to correctly interface with this data. Requiring the user to switch file systems departs from this mission. The proposed translation would bring this work back in line with the rest of the thesis.

   Additionally, if we proved such a translation correct, it would become easier to verify the correctness of the implementation, whether informally or formally.

   One could combine the two translations to run ZFS as a layer in between POSIX. In theory, this would allow users to run their applications as is, while retaining the serializability guarantee.

   Finally, having both of these translations would imply that the two interfaces have equal expressive power, which, while unsurprising, is an attractive notion.

3. Finally, the POSIX encoding only captures some pieces of POSIX (though we and others [15] would argue that they are the core pieces), so there is plenty of work to be done in extending the encoding. Three obvious candidates for extension are: (1) The many errors that we do not model; (2) the higher-level features that we do not model, like permissions, file attributes, etc; and (3) the plethora of commands that we do not model, many of which can simply be derived from those that we do provide. Most of this work should be straightforward, though every addition increases complexity.

For ZFS in general, there are also three key ideas that we have wanted to pursue:

1. One of the main reasons we embarked on this project was to solve issues that we were having with TxForest. For example, we wanted to offer guarantees with respect to arbitrary concurrent processes and we wanted to be able to reason about the whole chain of semantics, doing away with informally specified file systems. While ZFS supports this in theory, we still want to tightly integrate ZFS and TxForest, both semantically and practically so that we can see the touted benefits of this project.
2. Another key feature that we desire for ZFS is a Hoare-style logic for reasoning about correctness. Eventually, we would want to lift this reasoning all the way up to TxForest. There is a significant amount of work to be done to design such a logic, but reasoning about the correctness of programs in transactional file systems (or indeed, file systems in general), is an underexplored problem.
3. Finally, a significant advantage touted for domain-specific languages for ad hoc data processing, is the ability to get a typed representation of a snippet of your file system. Why not consider a typed file system? The reason that we left the data contained in files abstract in ZFS was to consider precisely this idea. If files did not

have to contain just strings, but instead could contain other typed data, it would be straightforward to view the entire file system as one, large, typed tree.

With this view, it seems plausible that we could improve the correctness of the file system interfacing code. A simple version would just allow (or require) users to specify the expected type of the data they are requesting. A more complex version could give types to entire traversals of the zipper and perhaps generate a very simple check to see if we will encounter any type errors given the current state of the file system. Due to ZFS' isolation guarantees, the current state of the file system will also be the state in which the program runs, unless it runs into a conflict, in which case it could retry if it is still well-typed, and otherwise abort with some message.

## 6.6 Conclusion

We have presented the Zipper File System (ZFS), a transactional file system with a novel underlying abstraction and a formal semantics describing its behavior. This file system is based on Kiselyov's ZFS design [27]. The zipper abstraction closely captures the usage of a file system, by neatly representing both the tree structure and the working path. It also allows a new, functional view of a file system, which, we expect, can allow for a simple and efficient implementation of the semantics.

We have designed a denotational semantics to describe the operation of ZFS, both at the local and global level. Additionally, we have designed an operational global semantics, which allows concurrent transactions, and proven that the semantics provides serializable transactions. We have designed and formalized a semantics for POSIX and a translation from POSIX to ZFS. In theory, this allows users to run their usual applications that use a POSIX interface on top of ZFS, getting serializable transactions without any additional work.

Finally, we briefly touched on our prototype implementation of ZFS in OCaml. We also described six ideas for potential extensions of different scope to both ZFS and the POSIX semantics.

Chapter 7

# Related Work

This chapter summarizes research that is closely related to the work described in this dissertation. I have split the related work into four categories: (1) Data processing languages; (2) Transactional file system; (3) POSIX semantics; and (4) Zippers.

## 7.1 Data Processing Languages

This section covers the data processing languages PADS (Processing Ad hoc Data Sources), Forest, Type Providers, and LINQ. Note that PADS and Forest have already been discussed in Sections 2.2 and 2.3.

PADS (**P**rocessing **A**d hoc **D**ata **S**ources) is a domain-specific language for processing ad hoc data. It differs from the Forest-derived work described in this dissertation largely with its focus on single files rather than filestores. Section 2.2 gives a more detailed description. The PADS line of work explored four branches of research, most of which are discussed in [10]. These branches of inquiry, and their suggested solutions, should also be applicable to the work in this dissertation:

**Data Description Calculus.** The first branch defined a calculus and semantics for data description languages [9, 30]. The key idea is to enable a comparison between data description languages—in the style of Landin's seminal work *The Next 700 Programming*

*Languages* [29]—by defining a higher-level language in which all the others can be interpreted.

**Generic Programming Toolkit.** The second branch designed a generic programming toolkit for PADS [6]. This toolkit allowed programmers to write tools for any PADS description rather than being reliant on those tools that others have designed. The statistical analysis tools as well as the *XML* and *JSON* translators, mentioned in various chapters of this dissertation, are examples of such description-agnostic tools.

**Learning.** The third branch focused on lowering the barrier to entry for using PADS [11]. Given the complexity of some data formats described in this dissertation, it should be unsurprising that writing declarative specifications for them can be difficult. This branch of work lead to inference tools that generate specifications from ad hoc data without programmer input. Rather than getting immediately usable specification, the authors suggest that the greatest success of this branch lay in generating a prototype specification that humans can more easily improve upon as compared to starting from scratch.

**Forest.** The last branch is Forest [7], which extended PADS to work on filestores rather than only single files. Besides the practical benefits of providing specifications at the filestore level, the main theoretical contributions of Forest were a semantics. The semantics of Forest satisfied the round-tripping laws from the bidirectional programming literature [12], assuring users that Forest updates filestores in a sensible manner.

Beyond the PADS line of work, to which this dissertation belongs, there are several other language-based approaches to data processing. Standard parser generators, like Yacc [26], provide facilities for parsing data, though they do not tend to allow data-dependent parsing, nor are they as convenient and all-in-one as PADS.

**Type Providers.** F# Type Providers [2] share with Forest the ability to provide typed representations for external data sources. However, the Type Providers lack a formal semantics and do not support all nested types (or specifications). Type Providers mostly facilitate use of existing data formats or data with a schema, but they do offer some support for creating type providers for custom data sources.

**LINQ.** C# LINQ [1] allows users to query data sources directly using a simple syntax. However, LINQ only accommodates data sources that support the `IEnumerable` interface. LINQ does not have a formal semantics and it cannot use declarative specifications of filestores to provide parsers into typed representations of data.

## 7.2 Transactional File Systems

Work in transactional file systems stretches across several decades. Unlike ZFS, all other transactional file systems that I am aware of lack a formal semantics and a full proof of their guarantees. Note that there are no commonly-used transactional file systems. Some examples of transactional file systems include:

**PerDiS File System.** PerDiS [14] is an early example of a transactional file system. It uses an optimistic concurrency control scheme to provide serializable transactions. Rather than abort any transaction, PerDiS attempts to automatically merge conflicting transactions. Since this is not possible in general, the system will write both versions to disk and notify users to resolve the issue. Given the throughput expectations that we have of most file systems these days, particularly those that are likely to benefit most from being transactional, involving the user in this way is unlikely to be a feasible solution.

**Amino.** The Amino file system [41] provides ACID semantics. In Amino, individual system calls are transactional and applications can define blocks of systems calls that should be grouped into a single transaction. Amino is backwards compatible with standard POSIX applications, though, without additional changes, the applications can only benefit from the fact that individual system calls are transactional.

**Warp Transactional Filesystem.** The Warp Transactional Filesystem [5] (WTF) is a distributed file system that uses a fast, transactional key-value store in its backend to provide serializable transactions. At a high-level, WTF only appends data to the disk, removing references (stored in its transactional key-value store) to the data instead of the data itself. When there are no references, the data can safely be garbage collected. A file is composed of a list of such references, detailing where its different parts are stored. Clever, automatic defragmentation and compaction, amortizes the costs of this partitioning.

**TxFS.** TxFS [22] is a transactional file system that provides ACID semantics to its transactions. It leverages existing journaling mechanisms in file systems to provide these guarantees in user space. By leveraging well-studied and tested mechanisms, their confidence in the correctness of their system is increased.

## 7.3 POSIX Semantics

While the specification for POSIX [24] remains informal, various projects have attempted to specify or give semantics to subsets of POSIX:

**SibylFS.** SibylFS is an executable, formal model of the expected behavior of several file systems, including POSIX [36]. It does not supply a readable semantics exactly, but instead characterizes all allowable traces. SibylFS can be used as a test oracle for

file system authors who want assurance that their file system conforms to the POSIX specification.

**Reasoning About POSIX.** Gardner, Ntzik, and others' work on reasoning about POSIX file systems, starting with local reasoning [15] and culminating in Ntzik's PhD dissertation [35], provides an axiomatic semantics for a subset of the POSIX file system interface. They use this semantics to design a variant of a separation logic to enable formal reasoning about file system applications. We were partially inspired by their choice of core commands.

**Executable Semantics of POSIX.** Greenberg and Blatt design an operational semantics for a POSIX shell called Smoosh [19]. They implement Smoosh and benchmark its performance and conformance to the POSIX specification. In doing so, they manage to find bugs in other shells, test suites, and in the POSIX specification. Given its focus on the POSIX shell rather than a small subset of its file system interface, their semantics is significantly more complex than ours.

## 7.4 Zippers

There is much work on zippers, though most of it is not relevant to how we apply them in file systems. Kiselyov's work on a zipper-based file system [27, 28], however, was our main inspiration for using zippers. Kiselyov provides an implementation of his system, which works similarly to ours, in Haskell. However, he does not provide a semantics for either his ZFS, or POSIX, and thus provides no translation between them. Additionally, his focus is on the use of delimited continuations to provide certain nice properties in his prototype file system. Our system does not use continuations (though it could and it is a perfectly reasonable choice).

Other relevant work on zippers include:

**Huet's Zippers.** The concept of zippers was first published by Huet [23]. Huet identifies what we would now call a tree zipper, and he suggests that the zipper must have been reinvented on numerous occasions due to the simplicity and elegance of the idea. The key goal of his work, beyond elucidating the elegance and simplicity of the structure, was to enable tree updates without relying on destructive mutation or requiring logarithmic complexity for each update.

**McBride's Derivatives.** Finally, because it is some of the coolest work I have ever seen, I would be remiss if I do not mention McBride's work on the correspondence between zippers and the structure that they capture [31]. In particular, he identifies the fact that the standard derivative operation from calculus (*e.g.* if $y = x^2$ then $\frac{dy}{dx} = 2x$), if applied to a regular type, produces the type of the focus for a zippered version of that type. This blew my mind.

Chapter 8

# Conclusion

In this dissertation, I tackle problems that arise in processing the vast amounts of ad hoc data in the world. I have designed two declarative, domain-specific languages for ad hoc data processing, focusing on the systems surrounding them and formalizations that ensure the properties that we desire. Additionally, to resolve concurrency issues that were unavoidable in such systems, I designed a transactional file system with a formal semantics and provably serializable transactions.

**Incremental Forest.** Chapter 3 introduced Incremental Forest (iForest), an extension of Forest with a delay construct. This construct allows users to selectively process data in their file system rather than loading or storing it all at once. The delay construct causes a series of ripple effects in the full design leading to, for example, a tree transformation language that we call skins. We also formalized a cost model to help users quantify the advantages of using delays in different situations. With this cost model, we proved that users would never suffer higher costs from introducing more delays.

**Transactional Forest.** Taking the idea of delays to their extreme, Chapter 5 introduced Transactional Forest (TxForest). This work redesigned the programming model of Forest by using a zipper as its underlying data structure and designing an API around this zipper. The zipper is closely related to the maximally delayed structure from iForest, ensuring

that users will incur minimal cost for their desired application. This zipper structure additionally lends itself to a simple logging scheme, allowing us to design and formalize a transaction manager with provably serializable transactions among TxForest processes.

**Zipper File System.** Unfortunately, it proved impossible to supply concurrency guarantees among arbitrary processes in a POSIX file system. Rather than use an existing transactional file system, all of which lack a strong formal semantics, Chapter 6 introduced a newly-designed file system based on the concept of zippers (similar to Kiselyov's ZFS [27]). The zipper concept lends itself to file systems, and its functional structure provides nice theoretical properties. We designed a formal semantics for the Zipper File System (ZFS) and proved that the semantics guarantees serializable transactions among arbitrary concurrent users. We additionally designed a formal semantics for a subset of POSIX. This semantics closely matches POSIX's informal specification, and we proved that POSIX could be implemented on top of ZFS. This allows standard applications written with a POSIX API to benefit from ZFS's transaction semantics.

## 8.1 Limitations

There are several limitations of the work in this dissertation. Most lie in the work's practical benefits, which I will touch on first: The project needs adoption, the implementation needs additional work, the programming model needs to be evaluated by real users, and the specifications can be exceedingly complex. There are also several theoretical limitations, including trade-offs in the semantics of storing, an inability to update comprehensions directly, and the partiality of the TxForest semantics.

I believe that what has been accomplished work could benefit many people, particularly computational scientists. But four issues need to be addressed:

- **Implementation**. The prototypes require significant engineering to become usable.

- **Adoption**. Aside from a project's use as a guideline to future projects, the utility of any project is limited by its adoption by its target community. There is a need for marketing, since one cannot use what one does not know about. There is also a need to design a self-sustainable ecosystem around TxForest. Finally, I believe that the use of OCaml as a host language is detrimental to TxForest's adoption. To combat this issue, we now have a prototype implementation of TxForest for Python.
- **User-friendliness**. We have little evidence that our core programming model is accessible its intended users. The TxForest model, where users traverse a specification zipper, is quite different from previous versions of Forest and may be more challenging to use. We need users and user studies both, to determine whether a redesign is necessary and how such a redesign should look.
- **Complexity of Specifications**. The key benefits of our work rely on users writing specifications for their data formats. Unfortunately, these specifications can be tedious and difficult to write, particularly for more-complex data formats. The fact that they are reusable allows the cost to be amortized by sharing specifications, though this sharing would require an ecosystem of users to be a proper solution. As seen in the Related Work chapter, PADS offers a solution based on automatically inferring specifications from data, but significant research is required to generate truly usable specifications in most situations.

There are also several theoretical limitations of my work:

- **Updating Comprehensions**. It would be nice to be able to add and remove elements directly to/from comprehensions in TxForest. Such an interface would be a more natural way of interacting with comprehensions than our current solution of manually adding or removing files in the correct place elsewhere in the specification. Unfortunately, since expressions are unrestricted, it is impossible

to reverse-engineer comprehension expressions in general. Therefore, we cannot determine how a modified comprehension should be represented in the file system. Currently, Forest, iForest, and TxForest side-step this issue by making expressions unable to directly query the file system. Instead, comprehensions depend on other directory specifications. We can hide this detail from the user by using heuristics to modify the file system and dependency checks to ensure correctness before persisting changes. But for full control, the user would have to change the directory manually. This is currently unavoidable, but dissatisfying.

- **Total TxForest**. The semantics of TxForest are partial, which means it is possible to get stuck by using the wrong command at the wrong time. As in many languages, this problem could be avoided by introducing a type system. However, several commands in TxForest depend on the state of the file system (*e.g.* `Down` and thus `Next` and `Prev`), which is, by its nature, not statically known. We chose to keep the commands simple and accept the partial semantics as is. Another option would be to design a semantics that could never block due to dynamic properties. For example, one could change `Down` to `Down_Or` and have it take another command as input. It would then go down if possible and execute its input command otherwise. This change would also allow a sound type system for TxForest.
- **Loading/Storing Dependencies**. We made a trade-off in the semantics of iForest and TxForest regarding how to load and store dependencies. In Forest, loading and storing dependencies is straightforward, since these functions are performed at the granularity of a filestore. In iForest, storing is especially complex. For loading, we chose to automatically force anything that the requested specification relied on. Another option is to fail if users had not loaded that specification beforehand. Additionally, we chose not to cache loaded dependencies to minimize memory usage. However, that means that such dependencies may be reloaded multiple

times. Perhaps the best trade-off, in hindsight, would be to have a special cache for the dependencies and to drop them first if more memory is required. This would require significantly more engineering effort.

In TxForest, we dealt with dependencies by encapsulating them as a zipper in a context. We introduced a run construct that is specifically used to execute commands and expressions on these contexts. We like this solution because, as in the rest of TxForest, users must be explicit about what data they load in their dependencies. However, for storing, users are required to manually traverse the zipper and ensure that any changes they make propagate to produce a consistent filestore. There are facilities to check the consistency of the current filestore, which can help in this endeavor, but the manual process of explicitly storing every dependency is tedious at best. On the other hand, there are no unsound heuristics used, so it is a theoretically elegant solution.

## 8.2 Future Work

This section presents opportunities for future work that I encourage others (and myself!) to consider. I have marked projects that I think would be useful in ***P***ractice, ***R***esearch related, or both with ***P***, ***R***, or ***P+R*** respectively.

- **TxForest Implementation Improvements - *P***. An improved implementation is critical for usability, but unlikely to lead to new research results. Beyond general improvements like bug fixes, documentation, and additional library features, there are two more significant and directed projects that I have in mind:

    1. **Generic Tool Framework - *P***. Many versions of PADS, as well as the initial version of Forest, had a generic tool-building framework that allowed users to construct specification-agnostic tools. Designing and implementing such a

framework for TxForest is integral for a vibrant ecosystem and wide adoption. I believe that this should not be as difficult as in previous versions due to the zipper structure.

2. **Practical Interfaces for TxForest - *P***. The standard interface for TxForest, as seen in the semantics in Chapter 5, is low level and difficult to write programs against, barring innumerable comments. However, it is a nice core language upon which to build more usable, derived interfaces.

   We have built an interface that tries to match the surface level specifications rather than the core specifications. As a refresher here is an example of both from Section 5.3.1:

   ```
   surface = directory {
     max is "max" :: file;
     students is [s :: students | s <- matches RE "[a-z]+[0-9]+"]
   }

   core = ⟨max:"max":: File, ⟨dir:Dir, [s :: students | s ∈ e]⟩⟩
     where e = filter (Run Fetch_Dir dir) "[a-z]+[0-9]+"
   ```

   We would prefer to allow users to interact with their specification zipper as though it encapsulated the surface specification directly, rather than just the compiled core specification. Unfortunately, while this interface normally "does the right thing," it currently uses a number of unsound heuristics. We additionally supply several higher-level functions, like `map` and `fold`, that can interact with arbitrary zippers.

   However, there is much to be done and any work in his direction would likely be a good first step on the road to a generic tool framework. Beyond a sound surface interface, we would ideally have a wealth of use cases to draw from to determine what higher-level functions are most critical.

- **TxForest Users - *P+R***. Improvements to the implementation of TxForest will benefit usability. However, significant work is required to design a system like

TxForest that its target audience (*e.g.* computational scientists) will have an easy time using. There are three main thrusts I would like to see explored in this vein:

1. **Case Studies - *P+R***. While we have attempted some case studies, like the SWAT study in Chapter 3, we could use many more to guide our work into a usable state. In particular, I would love to see longitudinal case studies in which the targeted group uses the tool, we capture their experiences, and then they continue using the tool and we check back.
2. **User Studies - *P+R***. This leads us to user studies in general. While we have confidence in the expressiveness of our tool and language, its usability is largely unexplored. Conducting user studies to evaluate how easy various aspects of our system are to use and learn would help us achieve a balance between usability and elegant, parsimonious semantics.
3. **Ecosystem - *P***. The ecosystem of a language is made up of its user community and the plethora of libraries, documentation, and developer tools that they create and share amongst themselves. Building an ecosystem is onerous and has no guarantee of success. However, there is nothing like having a significant user base to find the issues. Furthermore, specifications are reusable, as are generic tools. The complexity of designing these is more easily justified if they have higher value due to reaching a greater set of people.

- **Automated Specification Learning - *P+R***. One nice option to mitigate the issue of specification complexity has been partially explored for both PADS [11] and Forest [7]: automate some of the specification learning process. We would not expect to be able to do this perfectly, but generating a strong starting point for users should already significantly reduce the barrier to entry. I would like this specification inference tool to be an integrated procedure that can construct specifi-

cations at the level of PADS and Forest and combine them properly. These days, I would expect that we could achieve remarkable results through Machine Learning.

- **Restricted Expression Language - *P+R***. Earlier, I mentioned the difficulties of correctly updating comprehensions (*i.e.* supplying formally specified add/remove element commands for comprehensions). I have additionally mentioned the difficulties of storing with dependencies in general.

  The core decision from which both of these symptoms arise is the expressiveness of our expression language. It is convenient to allow arbitrary expressions from our host language, since now users can write code that is familiar to them. On the other hand, it makes it formally impossible to properly analyze due to Turing completeness. If we could instead construct a restricted expression language that admitted a finite analysis, we would be able to design a nice semantics for comprehension updates and allow store commands that properly propagate updates to the entire filestore to ensure that it remains consistent.

  There are several properties that we would like such a restricted language to have, beyond admitting easier reasoning: (1) It should be able to represent all filestores that we are aware of and can imagine users needing; (2) it should disallow file system-related side effects; and (3) it should be easy to use, ideally resembling a fairly standard programming language.

- **TxForest Type System - *P+R***. As mentioned in the Limitations section, the semantics of TxForest has a partiality problem. While this is true of the semantics of most programming languages, I believe that this causes particular difficulty in TxForest because of the low-level interface. In other languages, partiality is usually resolved with a sound type system, ensuring that well-typed programs do not get stuck. I believe that a type system is an excellent solution to this problem in TxForest as well, with one key caveat:

In practice, the type systems of other languages tend to ignore file system operations, instead relying on exceptions or the file system interface's built-in error codes. This is because the file system is an inherently dynamic object that is not statically available during code compilation.

In TxForest, *every* interesting command and expression relates to the file system. So how might we reconcile the dynamic nature of the file system with our desire for well-typed code? I believe that there are two plausible, mutually compatible, approaches that should both be explored:

The first is a fairly minor redesign of the semantics, which we discovered as we were starting to design a type system: Every operation that requires the file system to have particular properties can be modified to take another operation as input. This operation would be executed if the file system does not have the expected property. For example, consider the `Down` command. This command requires the current file system node to be a directory with at least one child. We could instead design a command `Down_Or` $c$, which works the same as `Down` except when the current file system node is not a directory with at least one child. In this case, it executes $c$ instead. With such a modification, along with similar changes to the other operations, we could design a sound type system for TxForest, which could ensure that no well-typed program ever gets stuck, independent of the file system.

However, this is not a completely practical solution, since users often want to know that their filestore conforms to its specification. If users just input operations that throw errors as the alternative, then there is little practical benefit to this semantics change. Additionally, users often have implicit beliefs about their filestore, which they have not bothered specifying, such as the minimum number of elements in a comprehension. This is where the second approach helps.

It is easy to construct a simple function or script that checks whether a particular

path in a file system corresponds to a specification. It is more difficult to construct a script that ensures that the path corresponds to the user's implicit beliefs. However, we can infer some of these beliefs from how they interface with the zipper. It should be straightforward to get a sort of dual of our type system, which, instead of providing soundness at the cost of a changed semantics, determines what the targeted subtree of the file system must look like for the program to be correct. Importantly, such a system should easily be able to produce a minimal set of requirements, which may be significantly cheaper to check than, nor be implied by, full specification conformance. We could then generate a script that, given a path, checks whether it has the minimally required shape for the program to run correctly.

- **Logic - *P+R***. As mentioned in the ZFS chapter, I would love a machine-checkable logic for reasoning about, and proving, the correctness of file system applications. However, there is no reason why this should stop at ZFS. Such a logic for TxForest would be useful with or without an accompanying logic for ZFS. Additionally, it should be possible to serve a dual function, much like the one mentioned in the previous paragraph, where users could include their desired post-conditions in the logic and we could both derive the necessary preconditions and generate a script to check whether they hold.
- **Various ZFS Improvements - *P+R***. Finally, there are a variety of other ZFS improvements mentioned in Chapter 6. These include providing a translation from ZFS to POSIX, proving the correctness of both translations, and exploring the possibilities of a typed file system.

## 8.3 Postlude

This, dear reader, is the final section of my dissertation, in which I have decided to provide a more biased view of my work. There are only two points that I wish to touch on here, pertaining to my conviction that this is a real-world, important problem, and my thoughts on the most impactful branches of future work.

First, I want to move away from the standard research spiel about why these problems and solutions are important. I think it's all true, of course, but beyond that, I truly, deeply believe that ad hoc data is a vast real-world problem. Further, I am constantly reminded of this fact when I speak to doctoral students in other fields and tell them about my work, because they frequently tell me that they are having exactly this problem. While I may not believe that they should use the current implementation of my system, I haven't seen a single idea that I think better addresses this problem. It is possible that there already exists a better solution than my work out there, but if so, no one (including me) seems to know about it.

For the future work, I attempted to provide a list that includes at least some project which is interesting and executable to anyone who might plausibly read this dissertation. However, I believe that certain lines of inquiry would have a larger (or smaller) impact than others:

1. I think that the **TxForest Users** line of work is absolutely essential. We need to design these systems in collaboration with our intended users and a thriving ecosystem would see the system improved enough and used enough to make a real difference.
2. I believe that designing a **Restricted Expression Language** would both be deeply fascinating work and offer significant leverage for improving the language.
3. I think that the **Logic** would be useful, given the current push for verified computa-

tion, but would require an exceptional design and tool for checking it.

4. The **Typed File System** sounds extremely cool to me and I like the elegance of a potential **TxForest Cost Model**, but I am dubious that either would turn out to be especially useful.

With that, I would like to thank you for reading my dissertation and tell you how impressed I am that you got this far. I hope you got something out of it and please feel free to contact me if you have questions about any of it!

# Bibliography


[1] 2017. Language-Integrated Query (LINQ) (C#). https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/concepts/linq/

[2] 2018. Type Providers - F#. https://docs.microsoft.com/en-us/dotnet/fsharp/tutorials/type-providers/

[3] Jonathan DiLorenzo, Katie Mancini, Kathleen Fisher, and Nate Foster. 2019. Tx-Forest: A DSL for Concurrent Filestores. In *APLAS 2019: Programming Languages and Systems*, Anthony Widjaja Lin (Ed.). Springer International Publishing, Cham, 332–354.

[4] Jonathan DiLorenzo, Richard Zhang, Erin Menzies, Kathleen Fisher, and Nate Foster. 2016. Incremental Forest: a DSL for efficiently managing filestores. In *ACM SIGPLAN Notices (OOPSLA)*, Vol. 51. 252–271.

[5] Robert Escriva and Emin Gün Sirer. 2016. The Design and Implementation of the Warp Transactional Filesystem. In *Proceedings of the 13th Usenix Conference on Networked Systems Design and Implementation (NSDI'16)*. USENIX Association, Berkeley, CA, USA, 469–483.

[6] Mary Fernández, Kathleen Fisher, J. Nathan Foster, Michael Greenberg, and Yitzhak Mandelbaum. 2008. A Generic Programming Toolkit for PADS/ML: First-Class

Upgrades for Third-Party Developers. In *Proceedings of the 10th International Conference on Practical Aspects of Declarative Languages (PADL'08)*. Springer-Verlag, Berlin, Heidelberg, 133–149.

[7] Kathleen Fisher, Nate Foster, David Walker, and Kenny Q. Zhu. 2011. Forest: A Language and Toolkit for Programming with Filestores. In *Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming (ICFP '11)*. ACM, New York, NY, USA, 292–306. https://doi.org/10.1145/2034773.2034814

[8] Kathleen Fisher and Robert Gruber. 2005. PADS: A Domain-specific Language for Processing Ad Hoc Data. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '05)*. ACM, New York, NY, USA, 295–304. https://doi.org/10.1145/1065010.1065046

[9] Kathleen Fisher, Yitzhak Mandelbaum, and David Walker. 2010. The next 700 Data Description Languages. *Journal of the ACM (JACM)* 57, 2 (2010), 1–51.

[10] Kathleen Fisher and David Walker. 2011. The PADS Project: An Overview. In *Proceedings of the 14th International Conference on Database Theory (ICDT '11)*. ACM, New York, NY, USA, 7. http://doi.acm.org/10.1145/1938551.1938556

[11] Kathleen Fisher, David Walker, Kenny Q. Zhu, and Peter White. 2008. From Dirt to Shovels: Fully Automatic Tool Generation from Ad Hoc Data. In *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '08)*. Association for Computing Machinery, New York, NY, USA, 421–434. https://doi.org/10.1145/1328438.1328488

[12] J. Nathan Foster, Michael B. Greenwald, Jonathan T. Moore, Benjamin C. Pierce, and Alan Schmitt. 2007. Combinators for Bidirectional Tree Transformations: A Linguistic Approach to the View Update Problem. *ACM Transactions on Program-*

*ming Languages and Systems (TOPLAS)* 29, 3 (May 2007). Short version in POPL '05.

[13] Mark Gabriel, Christopher Knightes, Ellen Cooter, and Robin Dennis. 2015. Evaluating relative sensitivity of SWAT-simulated nitrogen discharge to projected climate and land cover changes for two watersheds in North Carolina, USA. *Hydrological Processes* (2015).

[14] João Garcia, Paulo Ferreira, and Paulo Guedes. 1998. The PerDiS FS: A Transactional File System for a Distributed Persistent Store. In *Proceedings of the 8th ACM SIGOPS European Workshop on Support for Composing Distributed Applications (EW 8)*. ACM, New York, NY, USA, 189–194. https://doi.org/10.1145/319195.319224

[15] Philippa Gardner, Gian Ntzik, and Adam Wright. 2014. Local Reasoning for the POSIX File System. In *Proceedings of the 23rd European Symposium on Programming Languages and Systems - Volume 8410*. Springer-Verlag, Berlin, Heidelberg, 169–188. https://doi.org/10.1007/978-3-642-54833-8_10

[16] Kaushal K Garg, Luna Bharati, Anju Gaur, Biju George, Sreedhar Acharya, Kiran Jella, and B Narasimhan. 2012. Spatial mapping of agricultural water productivity using the SWAT model in Upper Bhima Catchment, India. *Irrigation and Drainage* 61, 1 (2012).

[17] Victor Gaydov. 2016. File locking in Linux. Retrieved April 4, 2020 from https://gavv.github.io/articles/file-locks/#mandatory-locking

[18] James N Gray. 1978. Notes on data base operating systems. In *Operating Systems*. Springer, 393–481.

[19] Michael Greenberg and Austin J. Blatt. 2019. Executable Formal Semantics for the POSIX Shell. *Proc. ACM Program. Lang.* 4, POPL, Article Article 43 (December 2019), 30 pages. https://doi.org/10.1145/3371111

[20] Björn Guse, Matthias Pfannerstill, and Nicola Fohrer. 2015. Dynamic Modelling of Land Use Change Impacts on Nitrate Loads in Rivers. *Environmental Processes* 2, 4 (2015).

[21] Theo Haerder and Andreas Reuter. 1983. Principles of Transaction-Oriented Database Recovery. *ACM Comput. Surv.* 15, 4 (December 1983), 287–317. https://doi.org/10.1145/289.291

[22] Yige Hu, Zhiting Zhu, Ian Neal, Youngjin Kwon, Tianyu Cheng, Vijay Chidambaram, and Emmett Witchel. 2019. TxFS: Leveraging File-System Crash Consistency to Provide ACID Transactions. *ACM Trans. Storage* 15, 2, Article 9 (May 2019), 20 pages. https://doi.org/10.1145/3318159

[23] Gérard Huet. 1997. The Zipper. *J. Funct. Program.* 7, 5 (September 1997), 549–554. https://doi.org/10.1017/S0956796897002864

[24] IEEE and The Open Group. 2018. The Open Group Base Specifications Issue 7, 2018 edition. Issue The Open Group Base Specifications Issue 7, 2018 edition. Retrieved April 28, 2020 from https://pubs.opengroup.org/onlinepubs/9699919799/

[25] IEEE and The Open Group. 2018. Utilities. Issue The Open Group Base Specifications Issue 7, 2018 edition. Retrieved April 4, 2020 from https://pubs.opengroup.org/onlinepubs/9699919799/idx/utilities.html

[26] Stephen C. Johnson and Ravi Sethi. 1990. *Yacc: A Parser Generator*. W. B. Saunders Company, USA, 347–374.

[27] Oleg Kiselyov. 2005. Tool demonstration: A zipper based file/operating system. In *Proceedings of the 2005 ACM SIGPLAN Workshop on Haskell (Haskell '05)*. http://okmij.org/ftp/continuations/ZFS/zfs-talk.pdf

[28] Oleg Kiselyov and Chung-chieh Shan. 2007. Delimited Continuations in Operating Systems. In *Modeling and Using Context (CONTEXT)*, Boicho Kokinov, Daniel C. Richardson, Thomas R. Roth-Berghofer, and Laure Vieu (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 291–302.

[29] P. J. Landin. 1966. The Next 700 Programming Languages. *Commun. ACM* 9, 3 (March 1966), 157–166. https://doi.org/10.1145/365230.365257

[30] Yitzhak H. Mandelbaum. 2006. *The theory and practice of data description*. Ph.D. Dissertation.

[31] Conor Mcbride. 2001. The Derivative of a Regular Type is its Type of One-Hole Contexts (Extended Abstract). http://strictlypositive.org/diff.pdf

[32] Daniel N Moriasi, Jeffrey G Arnold, Michael W Van Liew, Ronald L Bingner, R Daren Harmel, and Tamie L Veith. 2007. Model evaluation guidelines for systematic quantification of accuracy in watershed simulations. *Transactions of the ASABE* 50, 3 (2007).

[33] J.E. Nash and J.V. Sutcliffe. 1970. River flow forecasting through conceptual models part I — A discussion of principles. *Journal of Hydrology* 10, 3 (1970).

[34] Thanh Son Ngo, Duy Binh Nguyen, and Prasad Shrestha Rajendra. 2015. Effect of land use change on runoff and sediment yield in Da River Basin of Hoa Binh province, Northwest Vietnam. *Journal of Mountain Science* 12, 4 (2015).

[35] Gian Ntzik. 2017. *Reasoning About POSIX File Systems*. phdthesis. Imperial College London.

[36] Tom Ridge, David Sheets, Thomas Tuerk, Andrea Giugliano, Anil Madhavapeddy, and Peter Sewell. 2015. SibylFS: Formal Specification and Oracle-Based Testing for POSIX and Real-World File Systems. In *Proceedings of the 25th Symposium on Operating Systems Principles (SOSP '15)*. Association for Computing Machinery, New York, NY, USA, 38–53. https://doi.org/10.1145/2815400.2815411

[37] PADS Team. 2011. PADS Project. Retrieved March 21, 2020 from http://www.padsproj.org/

[38] Texas A&M University. 2012. SWAT Input/Output Documentation Version 2012. Retrieved March 21, 2020 from https://swat.tamu.edu/media/69296/swat-io-documentation-2012.pdf

[39] Texas A&M University. 2020. Soil and Water Assessment Tool. Retrieved March 21, 2020 from http://swat.tamu.edu

[40] Glynn Winskel. 1993. *The Formal Semantics of Programming Languages: An Introduction*. MIT Press, Cambridge, MA, USA.

[41] Charles P. Wright, Richard Spillane, Gopalan Sivathanu, and Erez Zadok. 2007. Extending ACID Semantics to the File System. *ACM Trans. Storage* 3, 2 (June 2007), 4–es. https://doi.org/10.1145/1242520.1242521

[42] Chris B. Zou, Lei Qiao, and Bradford P. Wilcox. 2015. Woodland expansion in central Oklahoma will significantly reduce streamflows – a modelling analysis. *Ecohydrology* (2015).

Chapter A

# Appendix to Chapter 3

## A.1 iForest Core Syntax

This section introduces a core calculus that we will use to prove a number of theorems about the language. The calculus extends the Forest core calculus [7] with a delay construct. Its syntax is as follows:

$$
\begin{array}{lll}
\text{Paths } p & ::= \cdot \mid p/u \\
\text{Contents } C & ::= \mathsf{File}\ u \mid \mathsf{Link}\ p \mid \mathsf{Dir}\ \ell \\
\text{File Systems } fs & ::= \{\!|\, p_1 \mapsto (a_1, C_1), \ldots, p_n \mapsto (a_n, C_n) \,|\!\} \\
\text{Specifications } s & ::= k^{\tau_2}_{\tau_1} \mid e :: s \mid \langle x{:}s_1, s_2 \rangle \mid [s \mid x \in e] \mid P(e) \mid s? \mid \mathit{Delay}(s)
\end{array}
$$

Meta-variable $u$ ranges over string constants, while meta-variable $a$ ranges over file system attributes (*e.g.*, the data returned when using the `stat` command: size, permissions, owner, time of last modification, *etc.*). The iForest surface language can be encoded into the core calculus as follows:

- $\mathit{File}$ and $\mathit{Link}$ are translated to constants $k^{\tau_2}_{\tau_1}$, where types $\tau_1$ and $\tau_2$ are appropriate representation and metadata types.
- $s$ `option` is translated to $s?$

- `dir` $\{\overline{x\ \texttt{is}\ s;}\}$ is translated to $\langle \_ : \mathsf{dir}^{unit}_{unit}, \langle x_1 : s_1, \langle x_2 : s_2, (\ldots \langle x_{n-1} : s_{n-1}, s_n \rangle) \rangle \rangle \rangle$, where the first component of the pair is a primitive that checks if the current node is a directory.
- $s$ `where` $e$ is translated to $\langle this : s, P(e) \rangle$. As in the surface language, this form gives the predicate expression access to the results of loading specification $s$ through the special variable $this$.
- $\langle s \rangle$ is translated to $Delay(s)$

The translations of other forms are straightforward.

## A.2 iForest Semantics

This section defines the formal semantics of iForest and proves several round-tripping properties.

Figure A.1 gives the representation and metadata types for each specification. The type $\tau$ `md` denotes the pair $(\texttt{bool} * \tau)$, where the boolean value indicates if an error occurred during loading. We write $\tau$ `cur` for the type of a cursor that returns $\tau$ when forced. Figures A.2 and A.3 define the load and store functions for each specification. Only the rules for delays (the last two rules in each column) differ from the semantics presented in the original Forest paper [7]. We explain the new rules in detail.

**Loading.** The judgment $E \vdash \texttt{load}\ (fs, p, s) \triangleright (r, md)$ holds if, in environment $E$, when we load file system $fs$ at path $p$ into memory as a specification $s$, we get a pair with representation $r$ and metadata $md$. We use a number of auxiliary functions. First, we use functions $load_k(E, fs, p)$ to implement the load function for each constant $k^{\tau_2}_{\tau_1}$. These functions take an environment $E$, a file system $fs$, and a path $p$ as arguments and return the representation and metadata of type $\tau_1$ and $\tau_2$ respectively. Second, the operator

$[\![e]\!]^E_\tau$ evaluates expression $e$ in environment $E$ and returns a value of type $\tau$. Third, we use standard projection functions $\pi_1$ and $\pi_2$.

The first new rule introduced in iForest is for loading a delayed specification. This rule returns as its representation a cursor (denoted $\mathbf{c}_{(E,p,s)}$) encapsulating the current environment, the path argument, and the specification that was delayed. This load always succeeds so the `unit md` that is returned is $(\texttt{true}, \texttt{()})$.

The second new rule uses a different judgment of the form $E' \vdash \texttt{load}_{\mathbf{c}}(\mathbf{c}_{(E,p,s)}, fs) \rhd (r, md)$. This judgment holds if, in environment $E'$, when we load cursor $\mathbf{c}_{(E,p,s)}$ in file system $fs$, we get representation and metadata pair $(r, md)$. The rule says that if we perform a normal load in the given file system, $fs$, with the environment, path, and specification encapsulated in the cursor, we will get the same result as when we perform a cursor-load in $fs$ in any environment.

**Storing.** The judgment $E \vdash \texttt{store}\ (fs, r, md, p, s) \rhd (fs', \varphi')$ holds if, in environment $E$ and file system $fs$, storing representation $r$ and metadata $md$ at path $p$ using specification $s$ produces a new file system $fs'$ and validator $\varphi'$.

A validator is a predicate on file systems that checks for internal inconsistencies in the representation and metadata [7]. We say that storing passes validation if the validator returns `true` when evaluated on the resulting file system. As with the load function, we use several auxiliary functions to define the store function. First, we use functions $store_k(E, fs, p, r, md)$ to implement the store function for each constant $k^{\tau_2}_{\tau_1}$. This function takes as arguments an environment $E$, a file system $fs$, a path $p$, a representation $r$ and metadata $md$. Second, we use an append operation on file systems, $fs_1 +\!\!+ fs_2$. Intuitively, this operation copies all contents from $fs_2$ to $fs_1$, overwriting any contents they have in common. Third, the $fs[p \mapsto \bot]$ function removes the mapping for a path $p$ in $fs$, or returns $fs$ if $p \notin dom(fs)$. Finally, the function $md^s_{default}$ computes "default" metadata for $s$.

| $s$ | $\mathcal{R}[\![s]\!]$ | $\mathcal{M}[\![s]\!]$ |
|---|---|---|
| $k_{\tau_1}^{\tau_2}$ | $\tau_1$ | $\tau_2$ `md` |
| $e :: s$ | $\mathcal{R}[\![s]\!]$ | $\mathcal{M}[\![s]\!]$ |
| $\langle x{:}s_1, s_2 \rangle$ | $\mathcal{R}[\![s_1]\!] * \mathcal{R}[\![s_2]\!]$ | $(\mathcal{M}[\![s_1]\!] * \mathcal{M}[\![s_2]\!])$ `md` |
| $[s \mid x \in e]$ | $\mathcal{R}[\![s]\!]$ `list` | $\mathcal{M}[\![s]\!]$ `list md` |
| $P(e)$ | `unit` | `unit md` |
| $s?$ | $\mathcal{R}[\![s]\!]$ `option` | $(\mathcal{M}[\![s]\!]$ `option`$)$ `md` |
| $Delay(s)$ | $(\mathcal{R}[\![s]\!] * \mathcal{M}[\![s]\!])$ `cur` | `unit md` |

**Figure A.1:** iForest representation and metadata type semantic functions

The first new storing rule for iForest says that storing a delayed specification returns an unchanged file system and a validator that always evaluates to `true`. The second new rule introduces a judgment of the form $E' \vdash \texttt{store}_{\mathbf{c}}(\mathbf{c}_{(E,p,s)}, fs, r, md) \rhd (fs', \varphi')$. This holds if, in environment $E'$, storing cursor $\mathbf{c}_{(E,p,s)}$ into file system $fs$ with representation $r$ and metadata $md$ yields a new file system and validator pair $(fs', \varphi')$. This rule says that such a store is equivalent to storing the given representation and metadata with the environment, path, and specification encapsulated in the cursor into the given file system.

**Round-tripping Properties.** The original Forest paper proved two round-tripping properties, showing that a load and a subsequent store causes no change to the file system (and passes validation) and that a store (if it passes validation) and a subsequent load gives back the same representation and metadata pair that was just stored. These properties also hold in iForest along with analogous properties for cursors. Formally:

**Theorem A.2.1** (LoadStore)**.** *Let $E$ be an environment, $fs$ and $fs'$ file systems, $p$ a path, $s$ a specification, $r$ a representation, $md$ metadata, and $\varphi'$ a validator. If*

$$E \vdash \texttt{load}\ (fs, p, s) \rhd (r, md)$$

$$E \vdash \texttt{store}\ (fs, r, md, p, s) \rhd (fs', \varphi')$$

$$\frac{}{E \vdash \texttt{load}\ (fs, p, k^{\tau_2}_{\tau_1}) \rhd load_k(E, fs, p)}$$

$$\frac{E \vdash \texttt{load}\ (fs, [\![p/e]\!]^E_{\texttt{filepath}}, s) \rhd (r, md)}{E \vdash \texttt{load}\ (fs, p, e :: s) \rhd (r, md)}$$

$$\frac{\begin{array}{c} E \vdash \texttt{load}\ (fs, p, s_1) \rhd (r_1, md_1) \\ (E, x \mapsto r_1, x_{md} \mapsto md_1) \vdash \texttt{load}\ (fs, p, s_2) \rhd (r_2, md_2) \\ b = ((\pi_1\ md_1) \wedge (\pi_1\ md_2)) \end{array}}{E \vdash \texttt{load}\ (fs, p, \langle x{:}s_1, s_2 \rangle) \rhd ((r_1, r_2), (b, (md_1, md_2)))}$$

$$\frac{\begin{array}{c} [\![e]\!]^E_{\alpha\ \texttt{list}} = [w_1, \ldots, w_n] \\ \forall i \in \{1, \ldots, n\}. (E, x \mapsto w_i) \vdash \texttt{load}\ (fs, p, s) \rhd (r_i, md_i) \\ b = \bigwedge_i^n \pi_1\ md_i \qquad rs = [r_1, \ldots, r_n] \qquad mds = [md_1, \ldots, md_n] \end{array}}{E \vdash \texttt{load}\ (fs, p, [s \mid x \in e]) \rhd (rs, (b, mds))}$$

$$\frac{b = [\![e]\!]^E_{\texttt{bool}}}{E \vdash \texttt{load}\ (fs, p, P(e)) \rhd (\texttt{()}, (b, \texttt{()}))}$$

$$\frac{p \in dom(fs) \qquad E \vdash \texttt{load}\ (fs, p, s) \rhd (r, md)}{E \vdash \texttt{load}\ (fs, p, s?) \rhd (\texttt{Some}(r), (\pi_1\ md, \texttt{Some}(md)))}$$

$$\frac{p \notin dom(fs)}{E \vdash \texttt{load}\ (fs, p, s?) \rhd (\texttt{None}, (\texttt{true}, \texttt{None}))}$$

$$\frac{}{E \vdash \texttt{load}\ (fs, p, Delay(s)) \rhd (\mathbf{c}_{(E,p,s)}, (\texttt{true}, \texttt{()}))}$$

$$\frac{E \vdash \texttt{load}\ (fs, p, s) \rhd (r, md)}{E' \vdash \texttt{load}_{\mathbf{c}}(\mathbf{c}_{(E,p,s)}, fs) \rhd (r, md)}$$

**Figure A.2:** iForest load function semantics

*then* $fs = fs'$ *and* $\varphi'(fs')$.

**Theorem A.2.2** (StoreLoad)**.** *Let* $E$ *be an environment,* $fs$ *and* $fs'$ *file systems,* $p$ *a path,* $s$ *a specification,* $r$ *and* $r'$ *representations,* $md$ *and* $md'$ *metadata, and* $\varphi'$ *a validator. If*

$$E \vdash \texttt{store}\ (fs, r, md, p, s) \rhd (fs', \varphi') \quad \varphi'(fs')$$

$$E \vdash \texttt{load}\ (fs', p, s) \rhd (r', md')$$

*then* $(r', md') = (r, md)$.

$$\frac{}{E \vdash \texttt{store}\ (fs, r, md, p, k^{\tau_2}_{\tau_1}) \rhd store_k(E, fs, p, r, md)}$$

$$\frac{E \vdash \texttt{store}\ (fs, r, md, [\![p/e]\!]^E_{\texttt{filepath}}, s) \rhd (fs', \varphi')}{E \vdash \texttt{store}\ (fs, r, md, p, e :: s) \rhd (fs', \varphi')}$$

$$\frac{\begin{array}{c} md = (b, (md_1, md_2)) \qquad r = (r_1, r_2) \\ E' = (E, x \mapsto r_1, x_{md} \mapsto md_1) \\ b' = (b = (\pi_1\ md_1) \wedge (\pi_1\ md_2)) \\ E \vdash \texttt{store}\ (fs, r_1, md_1, p, s_1) \rhd (fs_1, \varphi_1) \\ E' \vdash \texttt{store}\ (fs, r_2, md_2, p, s_2) \rhd (fs_2, \varphi_2) \\ \varphi' = (\lambda fs'.b' \wedge\ \varphi_1(fs') \wedge \varphi_2(fs')) \end{array}}{E \vdash \texttt{store}\ (fs, r, md, p, \langle x{:}s_1, s_2 \rangle) \rhd (fs_1 +\!\!+ fs_2, \varphi')}$$

$$\frac{\begin{array}{c} rs = [r_1, \ldots, r_j] \qquad mds = [md_1, \ldots, md_l] \\ [\![e]\!]^E_{\alpha\ \texttt{list}} = [w_1, \ldots, w_m] \qquad n = min(j, l, m) \\ b' = (b = \bigwedge_i^n \pi_1\ md_i) \qquad \forall i \in \{1, \ldots, n\}. \\ (E, x \mapsto w_i) \vdash \texttt{store}\ (fs, r_i, md_i, p, s) \rhd (fs_i, \varphi_i) \\ \varphi' = (\lambda fs'.(j = l = m) \wedge b' \wedge (\bigwedge_i^n \varphi_i(fs'))) \\ fs' = fs_1 +\!\!+ \ldots +\!\!+ fs_n \end{array}}{E \vdash \texttt{store}\ (fs, rs, (b, mds), p, [s \mid x \in e]) \rhd (fs', \varphi')}$$

$$\frac{\varphi' = \lambda fs'.b = [\![e]\!]^E_{\texttt{bool}}}{E \vdash \texttt{store}\ (fs, (), (b, ()), p, P(e)) \rhd (fs, \varphi')}$$

$$\frac{\begin{array}{c} E \vdash \texttt{store}\ (fs, r, md, p, s) \rhd (fs', \varphi') \\ \varphi_1 = (\lambda fs'.(b = \pi_1\ md) \wedge p \in dom(fs) \wedge \varphi'(fs')) \end{array}}{E \vdash \texttt{store}\ (fs, \texttt{Some}(r), (b, \texttt{Some}(md)), p, s?) \rhd (fs', \varphi_1)}$$

$$\frac{\varphi' = (\lambda fs'.md = \texttt{None} \wedge b \wedge p \notin dom(fs'))}{E \vdash \texttt{store}\ (fs, \texttt{None}, (b, md), p, s?) \rhd (fs[p \mapsto \bot], \varphi')}$$

$$\frac{\begin{array}{c} E \vdash \texttt{store}\ (fs, r, md^s_{default}, p, s) \rhd (fs', \varphi_1) \\ \varphi' = \lambda fs'.\texttt{false} \end{array}}{E \vdash \texttt{store}\ (fs, \texttt{Some}(r), (b, \texttt{None}), p, s?) \rhd (fs', \varphi')}$$

$$\frac{}{E \vdash \texttt{store}\ (fs, r, md, p, Delay(s)) \rhd (fs, \lambda fs'.\texttt{true})}$$

$$\frac{E \vdash \texttt{store}\ (fs, r, md, p, s) \rhd (fs', \varphi')}{E' \vdash \texttt{store}_{\mathbf{c}}(\mathbf{c}_{(E,p,s)}, fs, r, md) \rhd (fs', \varphi')}$$

**Figure A.3:** iForest store function semantics

**Theorem A.2.3** (iLoadStore)**.** *Let $E$, $E'$, and $E''$ be environments, $fs$ and $fs'$ file systems, $r$ a representation, $md$ a metadata, $\varphi'$ a validator, and $\mathbf{c}_{(E,p,s)}$ a cursor. If*

$$E' \vdash \texttt{load}_{\mathbf{c}}(\mathbf{c}_{(E,p,s)}, fs) \rhd (r, md)$$

$$E'' \vdash \texttt{store}_{\mathbf{c}}(\mathbf{c}_{(E,p,s)}, fs, r, md) \rhd (fs', \varphi')$$

*then $fs = fs'$ and $\varphi'(fs')$.*

**Theorem A.2.4** (iStoreLoad)**.** *Let $E$, $E'$, and $E''$ be environments, $fs$ and $fs'$ file systems, $r$ and $r'$ representations, $md$ and $md'$ metadata, $\varphi'$ a validator, and $\mathbf{c}_{(E,p,s)}$ a cursor. If*

$$E' \vdash \texttt{store}_{\mathbf{c}}(\mathbf{c}_{(E,p,s)}, fs, r, md) \rhd (fs', \varphi') \quad \varphi'(fs')$$

$$E'' \vdash \texttt{load}_{\mathbf{c}}(\mathbf{c}_{(E,p,s)}, fs') \rhd (r', md')$$

*then $(r', md') = (r, md)$.*

Note these judgments do not require the same environments since the environment in the cursor is used instead.

## A.3 Skin Core Syntax and Semantics

This section describes the formal syntax and semantics of skins and their accompanying type system.

**Syntax.** Figure A.4 defines the semantics for a skin core calculus. This syntax is a subset of the syntax from Figure 3.3. For example, we do not have $\rangle \langle$ or $map(h)$, which can be encoded using other constructs—*e.g.*, $\rangle \langle$ is equivalent to $\langle \rangle; \sim$.

The syntax of delay trees is similar to the syntax of iForest specifications. Delay trees are derived from specifications, but most details are stripped away, leaving only the basic structure and its delay annotations. This elision makes delay trees easier to work with in the formal semantics. Note that path expressions are eliminated in delay trees—we

found that they are almost never useful and reduced readability. Finally, we have the type syntax, which again mirrors specifications fairly closely, with a few additions: top ($\top$) and bottom ($\bot$) types, as well as intersections ($t_1 \wedge t_2$) and unions ($t_1 \vee t_2$).

**Semantics.** Next, we describe the semantics of the functions shown in Figures A.5, A.6, and A.7 and how they relate to iForest. Recall that we can apply a skin to a specification using the application construct $s@h$. Skin application can be evaluated in four steps, resulting in a new specification with the same underlying structure, but possibly different delay annotations:

1. Extract a delay tree $d$ from $s$ using $dtreeof$.
2. Check the type of $h$ against the type of $d$ by composing the $typeofH$ and $[\![\cdot]\!]_t$ functions.
3. If the preceding step produces a type error, report an error. Otherwise we apply the skin application function $[\![\cdot]\!]_h$ to $h$ and $d$ to generate $d'$.
4. Use the $apply$ function to apply the resulting delay tree $d'$ back to $s$ to generate $s'$, the final result of $s@h$.

The function $dtreeof$ strips away extraneous information from its argument to generate the corresponding delay tree. The function $typeofH$ computes the type of a skin. Since many skins can be applied to a variety of delay trees, we view types as sets to which delay trees (and, by extension, specifications) can belong. The function $typeofD$ computes a type from a delay tree. This function is not needed in iForest, but is used in a number of theorems.

The type of the primitive skins delay ($\langle\,\rangle$), negate ($\sim$), and identity (_) are all top ($\top$). This reflects the fact that these skins affect the delay annotation of a specification and do not depend on its structure. The structual skins for comprehensions ($[h]$), options ($h?$), and pairs ($\{h_1, h_2\}$) encode the corresponding constraints on the structure of the

delay tree and have the corresponding structural types (*i.e.*, $[t]$, $t?$, and $\{t_1, t_2\}$). The sequential composition skin ($h_1; h_2$) requires that the specification that it is applied to belong to the types of both of its sub-skins—*i.e.*, it has an intersection type ($t_1 \wedge t_2$). The union skin ($h_1 + h_2$) requires that the specification it is applied to must belong to either of the types of its sub-skins—*i.e.*, it has a union type ($t_1 \vee t_2$). Finally, the predicate skin ($h|t$) requires that the specification to belong to both the specified type and the type of its sub-skin.

The type-checking function ($[\![\cdot]\!]_t$) takes a type and a delay tree and checks whether the delay tree belongs to that type. The skin application function ($[\![\cdot]\!]_h$) applies a skin to a delay tree, producing a new delay tree with the same structure, but possibly different delay annotations. Note that skin application is partial—it is undefined if the delay tree does not belong to the type of the skin. However, since the type system is sound and complete, it is easy to ensure that the function will never be undefined in practice. The $apply$ function applies a delay tree to a specification, modifying its delay annotations, but not its structure. This function is partial because the structure of the delay tree and the specification must match. However, since delay trees are extracted from specifications (with $dtreeof$) and skins preserve structure (*cf*. Appendix A.4), partiality is not an issue in practice.

## A.4 Skin Properties and Theorems

This section presents the main lemmas and theorems tha we have proven about the skin language. Before we can prove these results, we need a few additional definitions. First, we define skin application formally:

**Definition A.4.1** (Skin Application)**.**

$$s@h \;=\; apply\ s\ ([\![h]\!]_h\ (dtreeof\ s))$$

Metavar Conventions

$s \in \textit{Specification}$
$h \in \textit{Skin}$
$d \in \textit{DTree}$
$t \in \textit{Type}$
$x \in \textit{Var}$
$e \in \textit{Exp}$
$b \in \mathbb{B}$

Function Types

$dtreeof : \textit{Specification} \to \textit{DTree}$
$typeofD : \textit{DTree} \to \textit{Type}$
$typeofH : \textit{Skin} \to \textit{Type}$
$[\![\cdot]\!]_t : \textit{Type} \to \textit{DTree} \to \mathbb{B}$
$[\![\cdot]\!]_h : \textit{Skin} \to \textit{DTree} \rightharpoonup \textit{DTree}$
$apply : \textit{Specification} \to \textit{DTree} \rightharpoonup \textit{Specification}$

Delay Tree Syntax

$$
\begin{array}{rl}
d ::= & k_{\tau_1}^{\tau_2} \\
| & d? \\
| & [d] \\
| & p \\
| & (d_1, d_2) \\
| & Delay(d)
\end{array}
$$

Skin Syntax

$$
\begin{array}{rl}
h ::= & \langle\ \rangle \\
| & \sim \\
| & \_ \\
| & h? \\
| & [h] \\
| & h|t \\
| & \{h_1, h_2\} \\
| & h_1 + h_2 \\
| & h_1; h_2
\end{array}
$$

Type Syntax

$$
\begin{array}{rl}
t ::= & cons_{\tau_1}^{\tau_2} \\
| & p \\
| & [t] \\
| & t? \\
| & \{t_1, t_2\} \\
| & t_1 \wedge t_2 \\
| & t_1 \vee t_2 \\
| & \top \\
| & \bot
\end{array}
$$

**Figure A.4:** Formal syntax of all components of the skin language

Next, we define skin equivalence:

**Definition A.4.2** (Skin Equivalence)**.**

$$h_1 = h_2 \iff$$

$$\forall d_0.\ ((\![\![h_1]\!]\!]_h\ d_0 = d \wedge\ [\![h_2]\!]_h\ d_0 = d') \implies d = d')$$

Hence, two skins are equivalent if and only if they produce the same result when applied to a given specification.

Using these definitions, we can prove a number of interesting equivalences on skins:

$$dtreeof : \textit{Specification} \rightarrow \textit{DTree}$$

$$\begin{array}{lll}
dtreeof & k^{\tau_2}_{\tau_1} & = k^{\tau_2}_{\tau_1} \\
dtreeof & e :: s & = dtreeof\ s \\
dtreeof & \langle x{:}s_1, s_2 \rangle & = (dtreeof\ s_1, dtreeof\ s_2) \\
dtreeof & [s \mid x \in e] & = [dtreeof\ s] \\
dtreeof & P(e) & = p \\
dtreeof & s? & = (dtreeof\ s)? \\
dtreeof & Delay(s) & = Delay(dtreeof\ s)
\end{array}$$

$$typeofD : \textit{DTree} \rightarrow \textit{Type}$$

$$\begin{array}{lll}
typeofD & k^{\tau_2}_{\tau_1} & = cons^{\tau_2}_{\tau_1} \\
typeofD & p & = p \\
typeofD & Delay(d) & = typeofD\ d \\
typeofD & [d] & = [typeofD\ d] \\
typeofD & d? & = (typeofD\ d)? \\
typeofD & (d_1, d_2) & = \{typeofD\ d_1, typeofD\ d_2\}
\end{array}$$

$$typeofH : \textit{Skin} \rightarrow \textit{Type}$$

$$\begin{array}{lll}
typeofH & \langle\ \rangle & = \top \\
typeofH & \sim & = \top \\
typeofH & \overline{\phantom{x}} & = \top \\
typeofH & [h] & = [typeofH\ h] \\
typeofH & h? & = (typeofH\ h)? \\
typeofH & \{h_1, h_2\} & = \{typeofH\ h_1, typeofH\ h_2\} \\
typeofH & h_1; h_2 & = (typeofH\ h_1) \wedge (typeofH\ h_2) \\
typeofH & h_1 + h_2 & = (typeofH\ h_1) \vee (typeofH\ h_2) \\
typeofH & h|t & = t \wedge (typeofH\ h)
\end{array}$$

**Figure A.5:** Formal semantics of skins: $dtreeof$, $typeofD$, and $typeofH$

$$[\![\cdot]\!]_t : \mathit{Type} \to \mathit{DTree} \to \mathbb{B}$$

```
[[t]]_t d =
  match d with
  | Delay(d) -> [[t]]_t d
  | d ->
    match (t,d) with
    | (cons^{T2}_{T1}, k^{T2}_{T1}) -> true
    | (p,p) -> true
    | (t?,d?) -> [[t]]_t d
    | ([t],[d]) -> [[t]]_t d
    | ({t1,t2},(d1,d2)) -> [[t1]]_t d1 && [[t2]]_t d2
    | (t1 ∧ t2,d) -> [[t1]]_t d && [[t2]]_t d
    | (t1 ∨ t2,d) -> [[t1]]_t d || [[t2]]_t d
    | (⊤,_) -> true
    | (⊥,_) -> false
    | _ -> false
```

$$[\![\cdot]\!]_h : \mathit{Skin} \to \mathit{DTree} \rightharpoonup \mathit{DTree}$$

```
[[h]]_h d =
  match h with
  | h|t -> if [[t]]_t d then [[h]]_h d
  | h1;h2 -> [[h2]]_h ([[h1]]_h d)
  | h1 + h2 ->
      if [[typeofH h]]_t d then [[h1]]_h d else [[h2]]_h d
  | h ->
    let d,del =
      match d with
      | Delay(d) -> d,true
      | d -> d,false
    in
    let d,del =
      match (h,d) with
      | (h?,d?) -> ([[h]]_h d)?,del
      | ([h],[d]) -> [[[h]]_h d],del
      | ({h1,h2},(d1,d2)) -> ([[h1]]_h d1,[[h2]]_h d2),del
      | (⟨ ⟩,d) -> d,true
      | (∼,d) -> d,(not del)
      | (_,d) -> d,del
    in
    if del
    then Delay(d)
    else d
```

**Figure A.6:** Formal semantics of skins: $[\![\cdot]\!]_t$ and $[\![\cdot]\!]_h$

$$apply : \textit{Specification} \to \textit{DTree} \rightharpoonup \textit{Specification}$$

```
apply s d =
  match d with
  | Delay(d) -> Delay(apply s d)
  | d ->
    let s =
      match s with
      | Delay(s) -> s
      | s -> s
    in
    match (s,d) with
    | (k_{τ1}^{τ2},k_{τ1}^{τ2}) -> k_{τ1}^{τ2}
    | (P(e),p) -> P(e)
    | (s?,d?) -> (apply s d)?
    | (e :: s,d) -> e :: (apply s d)
    | ([s | x ∈ e],[d]) -> [apply s d | x ∈ e]
    | (⟨x:s1, s2⟩,(d1, d2)) ->
        ⟨x:apply s1 d1, apply s2 d2⟩
```

**Figure A.7:** Formal semantics of skins: $apply$

**Theorem A.4.3** (Equalities on Skins)**.**

$$\sim;\sim \;=\; \_$$

$$h_1;(h_2;h_3) = (h_1;h_2);h_3$$

$$(h_1 + h_2);h_3 = h_1;h_3 + h_2;h_3$$

$$h_1;(h_2 + h_3) = h_1;h_2 + h_1;h_3$$

$$h;\_ = h = \_;h$$

Theorem A.4.3 states that double negation is the same as identity, and that composition is associative, distributes over union, and has the identity skin as a unit.

The next few lemmas are used to prove that skin application is compositional (Theorem A.4.8). First we show that application preserves delay trees:

**Lemma A.4.4** (Delay Tree Preservation)**.**

$$d \in dom(apply\ s) \implies dtreeof\ (apply\ s\ d) = d$$

Let $=_d^s$ denote equality of specifications modulo delay annotations. Two specifications are related by this operator if they have the same underlying structure. The next lemma tells us that the delay annotations in a specification are irrelevant to the result of the $apply$ function.

**Lemma A.4.5** (Apply Equivalence)**.**

$$s_1 =_d^s s_2 \implies apply\ s_1\ d = apply\ s_2\ d$$

Lemma A.4.6 shows that the $apply$ function does not change the structure of a specification, *i.e.*, the output is equivalent to the input modulo delays.

**Lemma A.4.6** (Apply Preservation)**.**

$$d \in dom(apply\ s) \implies apply\ s\ d =_d^s s$$

Lemma A.4.7 shows that applying $apply$ twice in sequence is equivalent to just the second application.

**Lemma A.4.7** (Apply Cancellation)**.**

$$d_1 \in dom(apply\ s) \implies$$

$$apply\ (apply\ s\ d_1)\ d_2 = apply\ s\ d_2$$

Finally, Theorem A.4.8 combines the previous four lemmas to prove that applying two skins to a specification one after another is equivalent to applying their composition.

**Theorem A.4.8** (Skin Composition)**.**

$$(s@h_1)@h_2 = s@h_1; h_2$$

Let $=_d^d$ denote equality of delay trees modulo delay annotations. The next two lemmas are used to prove that the type of a specification is not changed when a skin is applied (Theorem A.4.11). Lemma A.4.9 shows that the type of a delay tree is not affected by delay annotations.

**Lemma A.4.9** (Invariance under Delays)**.**

$$d_1 =_d^d d_2 \implies typeofD\ d_1 = typeofD\ d_2$$

Lemma A.4.10 shows that if two specifications are equivalent modulo delays (*i.e.*, have the same structure), then so are the delay trees extracted from them.

**Lemma A.4.10** (DTreeof Preservation)**.**

$$s_1 =_d^s s_2 \implies dtreeof\ s_1 =_d^d dtreeof\ s_2$$

Theorem A.4.11 shows that the type of a specification is not affected by skin application.

**Theorem A.4.11** (Invariance under Skin Application)**.**

$$dtreeof\ s \in dom([\![h]\!]_h) \implies$$

$$typeofD\ (dtreeof\ s) = typeofD\ (dtreeof\ s@h)$$

This property is important because it means that users only need to consider the base specification when writing a compatible skin. Any delays or skins applied to a specification or its sub-specifications are irrelevant.

The next three lemmas taken together show that the type system is sound and complete. Lemma A.4.12 shows that type checking only depends on the structure of a delay tree, not the delays.

**Lemma A.4.12** (Typing Invariance under Delays)**.**

$$(d_1 =_d^d d_2 \wedge [\![t]\!]_t\ d_1 \implies [\![t]\!]_t\ d_2)$$

Lemma A.4.13 shows that the underlying structure of a delay tree is not changed by skin application.

**Lemma A.4.13** (Skin Application Preservation)**.**

$$[\![h]\!]_h\ d_1 = d_2 \implies d_1 =_d^d d_2$$

Lemma A.4.14 shows that only the underlying structure of a delay tree determines whether or not a skin can be applied to it.

**Lemma A.4.14** (Domain Invariance under Delays)**.**

$$d_1 =_d^d d_2 \wedge d_1 \in dom([\![h]\!]_h) \implies d_2 \in dom([\![h]\!]_h)$$

Finally, using these lemmas, we prove that the type system is sound and complete.

**Theorem A.4.15** (Soundness)**.** *The type system is sound,* i.e.*,*

$$[\![typeofH\ h]\!]_t\ d \implies d \in dom([\![h]\!]_h)$$

**Theorem A.4.16** (Completeness)**.** *The type system is complete,* i.e.*,*

$$d \in dom([\![h]\!]_h) \implies [\![typeofH\ h]\!]_t\ d$$

Chapter B

# Appendix to Chapter 5

## B.1 Proofs

### B.1.1 Serializability

We restate and prove Theorem 5.4.1:

**Theorem 5.4.1** (Serializability)**.** *Let $FS, FS'$ be file systems, $GL, GL'$ be global logs, and $T$ a transaction pool such that $\forall t \in T.$ `initial` $FS$ $t$. Then:*

$$\langle FS, GL, T\rangle \rightarrow_G^* \langle FS', GL', \{\}\rangle \implies \exists \ell \in Perm(T).\ [\![\ell]\!]_{\mathbb{G}}\ FS = FS'$$

*where $\rightarrow_G^*$ is the reflexive, transitive closure of $\rightarrow_G$.*

*Proof.* The definitions of `initial` and `restart` are in Figure B.1. By the premise, Theorem B.1.4 and $T \setminus \{\} = T$, we have:

$$\exists [t_1; \ldots; t_n] \in Perm(T).$$
$$\langle \texttt{restart}\ FS\ t_1\rangle \xrightarrow{\sigma_1}{}_L^*\ \langle (E_1, p_1, ps_1, z_1), FS_1, \texttt{Skip}\rangle$$
$$\vdots$$
$$\wedge\ \langle \texttt{restart}\ FS_{n-1}\ t_n\rangle \xrightarrow{\sigma_n}{}_L^*\ \langle (E_n, p_n, ps_n, z_n), FS', \texttt{Skip}\rangle$$

```
initial fs (((({}, p, {}, z), fs, cs), (cs, ts, [])) when is_none? z.ancestor ≜ true
initial _ _ ≜ false

restart FS (((E, p, ps, z), fs, c), (cs, ts, σ)) ≜ (({}, p', {}, z'), FS, cs)
  where (p', z') = goto_root (E, p, ps, z) fs
```

**Figure B.1:** Definitions of `initial` and `restart`

By Lemma B.1.1 and `initial` $FS\ t$, we have:

$\langle \texttt{restart}\ fs\ t\rangle \xrightarrow{\sigma}{}^{*}_{L} \langle (E, p, ps, z), fs', \texttt{Skip}\rangle \implies [\![t]\!]_G\ fs = fs'$

Thus, we have:

$$\exists [t_1; \ldots; t_n] = \ell \in Perm(T).$$

$$[\![FS]\!]_G\ t_1 = FS_1 \ \wedge\ \ldots\ \wedge\ [\![FS_{n-1}]\!]_G\ t_n = FS'$$

$$\implies [\![\ell]\!]_{\mathbb{G}}\ FS = FS' \text{ (By definition of } [\![\cdot]\!]_{\mathbb{G}})$$

□

Lemma B.1.1 shows that running a transaction in the local operational semantics produces the same result as in the denotational semantics.

**Lemma B.1.1** (Operational to Denotational)**.** *Let $fs$ be a file system and $t = (((E, p, ps, z), \_, c), \_)$ be a transaction, then:*

$\langle (E, p, ps, z), fs, c\rangle \xrightarrow{\sigma}{}^{*}_{L} \langle (E', p', ps', z'), fs', \texttt{Skip}\rangle \implies [\![t]\!]_G\ fs = fs'$

*Proof.* By rule induction from a similar big-step semantics and the equivalence of the small-step and big-step semantics. See Winskel's book for a nearly identical proof [40].

□

**Definition B.1.2** (Well-formed Transactions)**.** A transaction $t$ is well-formed with respect to a file system $FS$ and a global log $GL$ (denoted $FS, GL \vdash t$) iff $t$ comes from running an initial transaction for some number of steps and $FS$ comes from merging the initial local file system of $t$ with the more recent parts of $GL$:

$$FS, GL \vdash t \iff$$
$$\exists fs, t'.\ \texttt{initial}\ fs\ t' \ \wedge\ \langle FS, GL, \{t'\}\rangle \rightarrow_G^* \langle FS, GL, \{t\}\rangle$$
$$\wedge\ GL' = \{le \mid le \in GL \ \wedge\ \texttt{get\_ts}\ le \ \geq\ \texttt{get\_ts}\ t\} \ \wedge\ FS = \texttt{merge}\ fs\ GL'$$

We call a transaction pool $T$ well-formed in a similar manner when every transaction in it is well-formed:

$FS, GL \vdash T \iff \forall t \in T.\ FS, GL \vdash t$

**Definition B.1.3** (Transaction Pool Difference)**.** Difference on transaction pools, written $T \setminus T'$, is defined when there is a file system and global log for which both transaction pools are well-formed. Then, transaction pool difference is exactly normal multiset difference where equality on elements is defined by having the same initial transaction, $t'$, as seen in Definition B.1.2.

The next theorem is amenable to induction and serves as a key component of our serializability theorem proof:

**Theorem B.1.4** (Inductive Serializability)**.** *Let $FS, FS'$ be file systems, $GL, GL'$ be global logs, and $T, T'$ transaction pools such that $FS, GL \vdash T$, then:*

$$\langle FS, GL, T\rangle \rightarrow_G^* \langle FS', GL', T'\rangle \implies$$
$$\exists [t_1; \ldots; t_n] \in Perm(T \setminus T').$$
$$\langle \texttt{restart}\ FS\ t_1\rangle \xrightarrow{\sigma_1}{}_L^* \langle (E_1, p_1, ps_1, z_1), FS_1, \texttt{Skip}\rangle$$
$$\vdots$$
$$\wedge\ \langle \texttt{restart}\ FS_{n-1}\ t_n\rangle \xrightarrow{\sigma_n}{}_L^* \langle (E_n, p_n, ps_n, z_n), FS', \texttt{Skip}\rangle$$

*Proof.* By induction on the multi-step relation $\rightarrow_G^*$. See Figure B.1 for a definition of `restart`. The reflexive case is straight-forward, while the transitive step relies on Lemma B.1.5 to be able to apply the induction hypothesis twice. The single-step case is significantly more complicated and entirely covered in Lemma B.1.6. □

**Lemma B.1.5** (Well-formedness Preservation)**.** *Let $FS, FS'$ be file systems, $GL, GL'$ be global logs, and $T, T'$ transaction pools, then:*

$FS, GL \vdash T \ \wedge \ \langle FS, GL, T\rangle \rightarrow_G^* \langle FS', GL', T'\rangle \implies FS', GL' \vdash T'$

*Proof.* By straightforward induction on the multi-step relation $\rightarrow_G^*$.

□

**Lemma B.1.6** (Single-step Serializability)**.** *Let $FS, FS'$ be file systems, $GL, GL'$ be global logs, and $T, T'$ transaction pools such that $FS, GL \vdash T$.*

*Then,* $\langle FS, GL, T\rangle \rightarrow_G \langle FS', GL', T'\rangle \ \wedge \ t \in (T \setminus T') \implies$

$\langle \texttt{restart}\ FS\ t\rangle \xrightarrow{\sigma}_L^* \langle (E', p', ps', z'), FS', \texttt{Skip}\rangle$

*Proof.* By induction on the single-step relation $\rightarrow_G$. The theorem holds vacuously unless the step is a commit. If the step is a commit, then it follows from Theorem B.1.7. □

**Theorem B.1.7** (Merge Property)**.** *Let $t = (((E, p, ps, z), fs, c), (cs, ts, \sigma))$ be a transaction, $GL$ a global log, and $FS$ a file system such that $FS, GL \vdash t$,* `check_log` $GL\ \sigma\ ts$*, and* `merge` $FS\ \sigma = FS_m$*.*

*Then,* $\langle \texttt{restart}\ FS\ t\rangle \xrightarrow{\sigma}_L^* \langle (E', p', ps', z'), FS_m, \texttt{Skip}\rangle$

*Proof.* Follows directly from Lemma B.1.13. We use $FS, GL \vdash t$ to conclude

$\exists t'.\ \langle FS, GL, \{t'\}\rangle \rightarrow_G^* \langle FS, GL, \{t\}\rangle$ and that $FS, GL \vdash t'$.

Then we can apply the lemma.

□

**Lemma B.1.8** (Partial Check Log)**.** *Let $ts$ be a timestamp, $GL$ a global log, and $\sigma$,$\sigma'$ local logs.*

*Then,* `check_log` $GL\ (\sigma \cdot \sigma')\ ts \implies$ `check_log` $GL\ \sigma\ ts \ \wedge$ `check_log` $GL\ \sigma'\ ts$

*Proof.* `extract_paths` $(\sigma \cdot \sigma') =$ `extract_paths` $\sigma \cup$ `extract_paths` $\sigma'$ □

**Lemma B.1.9** (Check Log Property)**.** *Let $t$ be a transaction with log and timestamp, $\sigma$ and $ts$, respectively, $GL$ a global log, and $FS$ a file system such that $FS, GL \vdash t$.*
*Then,* `check_log` $GL\ \sigma\ ts \implies FS \sim \sigma$

*Proof.* By induction on the structure of the log $\sigma$ and in the non-empty case, induction on the multi-step function $\rightarrow_G^*$ having used $FS, GL \vdash t$ to establish a derivation. See Figure B.2 for a definition of $\sim$.

Uses Lemma B.1.10 and Lemma B.1.12 as well as Lemma B.1.19 (in the transitive case).

**Lemma B.1.10** (Reads Correspond to Values)**.** *If* `Read` $C\ p \in$ `canonicalize` $\sigma$ *and* $\langle (E, p, ps, z), fs, c\rangle \xrightarrow{\sigma}{}_L^* \langle (E', p', ps', z'), fs', c'\rangle$.
*Then,* $fs(p) = C$.

*Proof.* By induction on the multi-step function $\xrightarrow{\sigma}{}_L^*$.

Uses Lemma B.1.11 in transitive case.

**Lemma B.1.11.** $\forall p.\ \nexists p'.$ `subpath` $p\ p' \ \wedge\ p' \in$ `writes` $\sigma$
$\wedge\ \langle (E, p, ps, z), fs, c\rangle \xrightarrow{\sigma}{}_L^* \langle (E'', p'', ps'', z''), fs'', c''\rangle \implies fs(p) = fs''(p)$

*Proof.* By induction on the multi-step function $\xrightarrow{\sigma}{}_L^*$.

□

□

**Lemma B.1.12** (Global to Local)**.** *Let $FS$ be a file system and $GL$ be a global log. Then,*
$\langle FS, GL, \{(((E, p, ps, z), fs, c), (cs, ts, \sigma))\}\rangle \rightarrow_G^*$
$\langle FS, GL, \{(((E', p', ps', z'), fs', c'), (cs, ts, \sigma \cdot \sigma'))\}\rangle$
$\wedge$ `check_log` $GL\ (\sigma \cdot \sigma')\ ts \implies$
$\langle (E, p, ps, z), fs, c\rangle \xrightarrow{\sigma'}{}_L^* \langle (E', p', ps', z'), fs', c'\rangle$

*Proof.* By induction on the multi-step relation $\rightarrow_G^*$. The transitive case is straightforward and relies on Lemma B.1.8. The reflexive case is trivial. In the step case, commit and restart are ruled out, leaving single transaction step. □

□

**Lemma B.1.13** (Merge Lemma)**.** *Let* $t = (td, (cs, ts, \sigma))$ *and* $t' = (td', (cs, ts, \sigma \cdot \sigma'))$ *be transactions,* $GL$ *a global log, and* $FS$ *a file system such that* $FS, GL \vdash t \wedge$ `check_log` $GL\ (\sigma \cdot \sigma')\ ts$.

*Then,*

$\langle FS, GL, \{t\}\rangle \rightarrow_G^* \langle FS, GL, \{t'\}\rangle \ \wedge$ `merge` $FS\ \sigma = FS' \ \wedge$ `merge` $FS'\ \sigma' = FS_m$

$\implies \langle$`insert` $FS'\ td\rangle \xrightarrow{\sigma'}{}_L^* \langle$`insert` $FS_m\ td'\rangle$

*Proof.* By induction on the multi-step relation $\rightarrow_G^*$. The transitive case gets the intermediate well-formedness and `check_log` results by relying on Lemma B.1.8 and Lemma B.1.5. Additionally, it relies on the fact that the global log monotonically grows at the same time as the file system changes, which means that the intermediate steps have the same $FS$ and $GL$.

The single-step case first rules out commit (because a transaction remains) and restart (because `check_log` $GL\ \sigma\ ts$). With only the local step case remaining, we induct on the single-step relation $\xrightarrow{\sigma'}{}_L$.

The IMP rules are straightforward, and mostly do not affect the file system. In the Forest Command case, we use Lemma B.1.14 and Lemma B.1.9 to derive $FS' \sim \sigma'$, then use Lemma B.1.15 and Lemma B.1.16 for Forest Navigations and Forest Updates respectively. Since Forest Navigations only produce reads (by Lemma B.1.15), we also note that $FS_m = FS'$ in these cases. □

**Lemma B.1.14** (Intermediate Well-formedness)**.** *Let* $t = (td, (cs, ts, \sigma))$ *and* $t' = (td', (cs, ts, \sigma \cdot \sigma'))$ *be transactions,* $GL$ *a global log, and* $FS$ *a file system.*

*Then,*

$\langle FS, GL, \{t\}\rangle \rightarrow_G^* \langle FS, GL, \{t'\}\rangle \ \wedge \ FS, GL \vdash t \ \wedge \ \texttt{merge}\ FS\ \sigma = FS_m \implies FS_m, GL \cdot (\texttt{add\_ts}\ ts\ \sigma) \vdash t'$

*Proof.* Follows from Definition B.1.2. □

**Lemma B.1.15** (Merge: Forest Navigations)**.** *If $FS \sim \sigma$, then*

$\forall fn.\ (\exists fs.\ [\![fn]\!]_c\ (E, p, ps, z)\ fs = (ctxt, fs, \sigma))$

$\implies \texttt{reads}\ \sigma = \sigma \ \wedge \ [\![fn]\!]_c\ (E, p, ps, z)\ FS = (ctxt, FS, \sigma)$

*Proof.* By induction on the Forest Navigations $fn$ and mutually dependent on Lemma B.1.17. Uses Lemma B.1.18 and Lemma B.1.20 to be able to apply Lemma B.1.17 and the induction hypothesis in sequence.

□

**Lemma B.1.16** (Merge: Forest Updates)**.** *If $FS \sim \sigma$, then*

$\forall fu.\ (\exists fs.\ [\![fu]\!]_c\ (E, p, ps, z)\ fs = (ctxt, fs', \sigma) \ \wedge \ \texttt{merge}\ FS\ \sigma = FS_m)$

$\implies [\![fu]\!]_c\ (E, p, ps, z)\ FS = (ctxt, FS_m, \sigma)$

*Proof.* By induction on the Forest Updates $fu$. Uses Lemma B.1.17 for subexpressions and Lemma B.1.18 and Lemma B.1.20 to focus on the write portions of the log. □

**Lemma B.1.17** (Merge: Expressions)**.** *If $FS \sim \sigma$, then*

$\forall e.\ (\exists fs.\ [\![e]\!]_e\ (E, p, ps, z)\ fs = (v, \sigma))$

$\implies \texttt{reads}\ \sigma = \sigma \ \wedge \ [\![e]\!]_e\ (E, p, ps, z)\ FS = (v, \sigma)$

*Proof.* By induction on the expressions $e$ and mutually dependent on Lemma B.1.15. For `Verify`, there is a further induction on $s$. For `Run` $fe\ e$, we apply the induction hypothesis twice and for `Run` $fn\ e$, we apply Lemma B.1.15 once and the induction hypothesis once. In both cases, we rely on Lemma B.1.18 and Lemma B.1.20.

□

```
FS ∼ σ ≜ ∀ Read C p ∈ canonicalize σ. FS(p) = C
canonicalize σ ≜ fold [] σ necessary

necessary acc (Read C p) ≜
  if subpath p (writes acc) ∨ p ∈ reads acc
  then acc
  else (Read C p) · acc
necessary acc (Write_file C1 C2 p) ≜
  (PathWritten p) · (necessary acc (Read C1 p))
necessary acc (Write_dir (File u) (Dir ℓ) p) ≜
  (PathWritten p) · (necessary acc (Read (File u) p))
necessary acc (Write_dir (Dir ℓ') (Dir ℓ) p) ≜
  fold (necessary acc (Read (Dir ℓ') p))
    ((ℓ \ ℓ') ∪ (ℓ' \ ℓ))
    (λacc u. PathWritten p/u · acc)
```

**Figure B.2:** Log Compatibility definition

**Lemma B.1.18** (Log Compatibility with Reads)**.** $FS \sim (\sigma \cdot \sigma') \wedge \mathit{reads}\ \sigma = \sigma \implies FS \sim \sigma \wedge FS \sim \sigma'$

*Proof.* The first part follows from Lemma B.1.20. The second from the fact that $\mathtt{canonicalize}\ \sigma' \subseteq \mathtt{canonicalize}\ ((\mathtt{reads}\ \sigma) \cdot \sigma')$. □

**Lemma B.1.19** (Log Compatibility Combination)**.** $FS \sim \sigma \wedge FS \sim \sigma' \implies FS \sim (\sigma \cdot \sigma')$

*Proof.* $\mathtt{reads}\ (\mathtt{canonicalize}\ (\sigma \cdot \sigma')) \subseteq \mathtt{reads}\ (\mathtt{canonicalize}\ \sigma) \cup \mathtt{reads}\ (\mathtt{canonicalize}\ \sigma')$ □

**Lemma B.1.20** (Log Compatibility Parts)**.** $FS \sim (\sigma \cdot \sigma') \implies FS \sim \sigma$

*Proof.* $\mathtt{reads}\ (\mathtt{canonicalize}\ \sigma) \subseteq \mathtt{reads}\ (\mathtt{canonicalize}\ (\sigma \cdot \sigma'))$ □

## B.1.2 Properties

**Consistency.** We restate the consistency theorems from Section 5.3 and give the main idea of their proofs.

**Theorem 5.3.1.** *Consistency implies partial consistency:*

$$\forall ps.\ \texttt{consistent?}\ (\texttt{Consistent}\ (p, ps, z)\ fs) \implies$$
$$\texttt{consistent?}\ (\texttt{PConsistent}\ (p, ps, z)\ fs)$$

**Theorem 5.3.2.** *Partial Consistency is monotonic w.r.t. the path set:*

$$\forall ps_1, ps_2.\ ps_2 \subseteq ps_1 \implies$$
$$\texttt{consistent?}\ (\texttt{PConsistent}\ (p, ps_1, z)\ fs) \implies$$
$$\texttt{consistent?}\ (\texttt{PConsistent}\ (p, ps_2, z)\ fs)$$
$$\wedge\ \texttt{complete?}\ (\texttt{PConsistent}\ (p, ps_2, z)\ fs) \implies$$
$$\texttt{complete?}\ (\texttt{PConsistent}\ (p, ps_1, z)\ fs)$$

**Theorem 5.3.3.** *Given a zipper $z$ and a path set $ps'$ that covers the entirety of $z$, partial consistency holds iff full consistency holds:*

$$\forall ps, ps'.\ \texttt{Cover}\ (p, ps', z)\ fs\ \wedge\ ps' \subseteq ps \implies$$
$$\texttt{consistent?}\ (\texttt{Consistent}\ (p, ps, z)\ fs) \iff$$
$$\texttt{consistent?}\ (\texttt{PConsistent}\ (p, ps, z)\ fs)$$

*Proof.* The proofs of these three theorems are straightforward by induction on the structure of the specification in $z$. The theorems ignore the log portion of partial consistency. □

**Core Calculus Equivalences.** We present several equivalences in the core calculus.

**Definition B.1.21** (Equivalence modulo logs)**.** We define equivalence modulo logs inductively as follows. We consider partial functions to be total with unmapped values mapping to $\bot$:

$$\bot \equiv \_$$
$$\_ \equiv \bot$$
$$((E, p, ps, z), fs, \_) \equiv ((E, p, ps, z), fs, \_)$$
$$f \equiv f' \qquad \text{when } \forall v.\ f\ v \equiv f'\ v$$

**Lemma B.1.22** (Core Calculus Equivalences)**.**

$$[\![\texttt{Down};\texttt{Up}]\!]_c \equiv [\![\texttt{Skip}]\!]_c$$

$$[\![\texttt{Into\_Opt};\texttt{Out}]\!]_c \equiv [\![\texttt{Skip}]\!]_c$$

$$[\![\texttt{Into\_Comp};\texttt{Out}]\!]_c \equiv [\![\texttt{Skip}]\!]_c$$

$$[\![\texttt{Into\_Pair};\texttt{Out}]\!]_c \equiv [\![\texttt{Skip}]\!]_c$$

$$[\![\texttt{Next};\texttt{Prev}]\!]_c \equiv [\![\texttt{Skip}]\!]_c$$

$$[\![\texttt{Prev};\texttt{Next}]\!]_c \equiv [\![\texttt{Skip}]\!]_c$$

**Round-Tripping Laws.** We present several round-tripping laws in the style of lenses [12].

**Lemma B.1.23** (Round-Tripping Laws)**.**

$$[\![\texttt{Store\_File Fetch\_File}]\!]_c \equiv [\![\texttt{Skip}]\!]_c \qquad \textit{File-Load-Store}$$

$$[\![\texttt{Store\_Dir Fetch\_Dir}]\!]_c \equiv [\![\texttt{Skip}]\!]_c \qquad \textit{Dir-Load-Store}$$

$$[\![\texttt{Store\_File } u_1;\texttt{Store\_File } u_2]\!]_c \equiv [\![\texttt{Store\_File } u_2]\!]_c \qquad \textit{File-Store-Store}$$

$$[\![\texttt{Create\_Path};\texttt{Create\_Path}]\!]_c \equiv [\![\texttt{Create\_Path}]\!]_c \qquad \textit{CreatePath-Store-Store}$$

$$[\![x := u;\texttt{Store\_File } u;x := \texttt{Fetch\_File}]\!]_c \equiv [\![x := u]\!]_c \qquad \textit{File-Store-Load}$$

$$[\![x := \ell;\texttt{Store\_Dir } \ell;x := \texttt{Fetch\_Dir}]\!]_c \equiv [\![x := \ell]\!]_c \qquad \textit{Dir-Store-Load}$$

Note that $[\![\texttt{Store\_Dir } \ell_1;\texttt{Store\_Dir } \ell_2]\!]_c \equiv [\![\texttt{Store\_Dir } \ell_2]\!]_c$ is conspicuously missing. In fact, it does not hold. Consider the situation where $\ell_2$ is the current contents of the given directory. In this case, `Store_Dir` $\ell_2$ is a no-op, and thus the right-hand side is equivalent to `Skip`. However, if, for example, $\ell_1 = []$, then the left-hand side will turn every child into File $\varepsilon$.