Incremental Forest: A DSL for Efficiently Managing Filestores

Jonathan DiLorenzo
Cornell University, USA
dilorenzo@cs.cornell.edu

Richard Zhang *
University of Pennsylvania, USA
rmzhang@seas.upenn.edu

Erin Menzies
Cornell University, USA
egm55@cornell.edu

Kathleen Fisher
Tufts University, USA
kfisher@eecs.tufts.edu

Nate Foster
Cornell University, USA
jnfoster@cs.cornell.edu

Abstract

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 runtime system could load gigabytes of data, even if only a few bytes were needed. This paper 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 the soundness and completeness of the type system and a variety of algebraic properties of skins. We describe an OCaml implementation and evaluate its performance on applications developed in collaboration with watershed hydrologists.

* Work performed at Cornell University.

OOPSLA'16, November 2–4, 2016, Amsterdam, Netherlands

http://dx.doi.org/10.1145/2983990.2984034

Categories and Subject Descriptors D.3.2 [*Programming Languages*]: Language Classifications—Specialized application languages

Keywords Data description languages, file systems, filestores, domain-specific languages, ad hoc data, laziness

1. Introduction

File systems are a popular way of storing persistent application data. Programmers choose to use file systems instead of traditional solutions such as key-value stores or relational databases for a variety of reasons. File systems are ubiquitous, being bundled with every major operating system. They have a low barrier to entry, since programmers can manipulate data directly using standard APIs and command-line tools. They do not impose overheads such as setting up special user accounts, configuring access control, defining schemas, creating tables, importing data, etc. They offer portability, since data is not "locked in" to a proprietary format and can be easily transferred from one system to another.

At the same time, file systems have a number of limitations that create practical hurdles in applications. Having to write code to traverse directories and parse file contents is tedious and error-prone. Even simple tasks, such as computing the number of entries within a given date range on a directory of server logs, requires opening, loading, and processing a large amount of data. In addition, APIs such as POSIX do not provide constructs for documenting assumptions about the file system. Applications that depend on certain files being present or directories being structured in particular ways lack mechanisms for declaring and enforcing those constraints. Simple mistakes such as a misnamed directory or a file with the wrong permissions can lead to application-level errors, but are difficult to detect and diagnose using existing tools.

Previous work on Forest [4] 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 solves the two main issues discussed above: it 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 precisely 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 obviously actually 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 paper 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, henceforth called Classic Forest when appropriate to avoid confusion, 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 undelayed 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 can be many ways to add delays to a given Forest specification. With no delays, the application can manipulate values stored on the file system directly, as in Classic 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 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) [13], 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 paper 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 paper is structured as follows. Section 2 motivates iForest's design and presents an overview of its main features. Section 3 formalizes Classic Forest. Section 4 presents the design of iForest and defines the syntax and semantics of the language. Section 5 introduces a dynamic cost model. Section 6 presents the skin language. Section 7 describes our experiences building various applications in iForest as well as quantitative experiments on SWAT data. We discuss related work in Section 8 and conclude in Section 9. The appendix formalizes the main features of iForest in a core calculus and presents theorems and proofs.

2. Overview

This section motivates the design of iForest using a simple running example. Consider the file system fragment shown in Figure 1. The root directory has two entries: an index file (`/index.txt`) and a data directory (`/data`). The di-

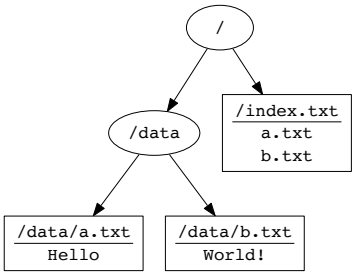


Figure 1. Example directory.

rectory contains a number of text files (`/data/a.txt` and `/data/b.txt`). There is a dependency between the index and the directory—the former should contain the names of the files in the latter.

OCaml Implementation. Suppose we wish to use this data as a simple filestore without using Forest. The first step would be to identify a type to represent the data in memory and a function to load data from the file system. For example, we might use a record type with one field for the index and another for the data, written here in OCaml:

```
type rep = { index:string; data:string list }
```

Next we can write the `load` function itself:

```
let load () : rep =
  let file_names = read_lines "/index.txt" in
  { index = String.concat "\n" file_names;
    data = List.map
             (fun f -> read_file ("data/" ^ f))
             file_names }
```

The helper function `read_file` uses the POSIX API to load the `string` contents of a single file, while `read_lines` returns a list of strings, one for each line in the file. Running this function yields the desired result:

```
{ index = "a.txt\nb.txt";
  data = ["Hello"; "World!"] }
```

However, in addition to being tedious for the programmer, there are several issues with this implementation. First, it does not handle all of the error conditions that could arise in practice—e.g., if one of the files in the `/data` directory were missing, the function would raise an exception rather than returning a partial result and an instructive error message. Second, it does not actually implement the complete functionality of a filestore. To finish the job, we would also need to write a `store` function that maps a possibly modified `rep` value back to the file system. Note that these issues arise even with a toy example! They are further exacerbated in the more complicated scenarios that arise in practice.

Classic Forest Implementation. Forest [4] 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. Returning to our running example, we could use the following Forest specification to specify the structure of the filestore:

```
d = dir {
  index is "index.txt" :: file;
  data is [ "data" :: f :: file
          | f <- $lines index$ ] }
```

To a first approximation, the specification `d` can be thought of as a type that specifies the expected structure and contents of a file system at a given path. The directory construct (`dir {...}`) 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 `index` and `data` that are associated with the representations of the nested specifications. The path construct (`"index.txt :: file`)[1] navigates to the specified path (`index.txt`), while the file primitive (`file`) specifies that the node at that path should be a file. The representation for a path is the representation for its nested specification, while the representation for a generic file is a `string`. A comprehension specifies a collection of values—in this case, the files at paths given in `index`. The notation `$...$` denotes that the enclosed code comes from the host language. The representation for a comprehension is a list. Note the dependency between the two parts of the specification—a situation that often arises in practice.

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

- A type `d_rep` for the in-memory representation:

  ```
  type d_rep = { index:string; data:string list }
  ```

- A type `d_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 d_md =
    { index_md:unit md;
      data_md:((unit md) list) md }
  ```

- Functions `d_load` and `d_store` that map between the on-disk and in-memory representations:

  ```
  val d_load : filepath -> d_rep * d_md
  val d_store : filepath -> d_rep * d_md -> unit
  ```

Unlike the manual implementation described previously, these functions automatically check for errors and internal

[1] The `::` operator associates to the right.

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 prints out the name and size of each file in the data directory:

```
let (rep,md) = d_load path in
if md.num_errors = 0 then
  let go f s = printf "%s: %d\n" f (length s) in
  List.iter2 go
    (lines rep.index) rep.data
else
  let error = String.concat "\n" md.error_msg in
  failwith (sprintf "%s" error)
```

When executed on the same file system as before, we get the following output:

```
a.txt: 5
b.txt: 6
```

In addition, the implementation provides a way to gracefully handle errors. For example, if the first file were deleted, creating an internal inconsistency in the data, we would get the following output,

```
Failure "/data/a.txt: no such file"
```

rather than a confusing run-time exception.

In addition to the basic features used in this simple example, Classic Forest offers a number of other constructs that are useful for describing real-world filestores. Options (`d option`) specify that `d` may either be present or absent. The representation is an OCaml `option` value. Predicates (`d where $e$`) specify that the constraints embodied in expression `e` must be satisfied. The representation is a `unit` value, but the metadata records an error if the constraints are not satisfied. Forest also supports recursive specifications. Finally, because the language is based on a compositional design it is relatively straightforward to add other operators.

Classic Forest Limitations. Classic 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 Classic Forest does not provide a way to make such tradeoffs.

Incremental Forest. iForest is an extension to Classic 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 file in the data directory by wrapping `file` with angle brackets, iForest's notation for the delay construct (shaded here in gray):

```
d1 = dir {
  index is "index.txt" :: file;
  data is [ "data" :: f :: <file>
          | f <- $lines index$ ] }
```

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

```
type d1_rep =
 { index:string;
   data:((string, unit md) cursor) list }
```

This `d_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:

```
d1_new path >>=
load >>= fun (r, md) ->
if md.num_errors = 0 then
  let go f cur acc =
    load cur >>= fun (s,_) ->
    printf "%s: %d\n" f (length s);
    acc
  in
  List.fold_right2 go (lines r.index)
                      r.data (return ())
else
  let error = String.concat "\n" md.error_msg in
  failwith (Printf.sprintf "%s" error)
```

This program is 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` function that takes a cursor as an argument. We create a new cursor using the function `d_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 standard monadic bind operation (>>=) sequences computations.

Now, rather than loading all data files 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 data file in memory, and the garbage collector could reclaim the memory for previously-loaded files. We could even avoid loading certain files entirely—e.g., those satisfying some predicate—by wrapping the `go` function above in a conditional—which would have significant performance benefits.

Skins. In general, there can be many different ways of adding delays to a Forest 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 Forest 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. Classic Forest, Formally

This section briefly reviews the syntax and semantics of Classic Forest to set the stage for iForest, which is described in the next section. Figure 2 (a) gives the syntax of the language. We assume a set of variables x and expressions e, and we write $\bar{t}$ as shorthand for the non-empty sequence $t_1, \ldots, t_n$, where t is any syntactic object. Meta-variable s ranges over specifications.

- *Files and Links.* Specs `file` and `link` describe filestores with a file and link, respectively, at the current path.
- *Paths.* Specs $e :: s$ describe filestores where expression e evaluates to a path that leads to contents described by s. Note that e may contain variables, which is useful in dependent specifications, as described below.
- *Options.* Specs s `option` describe filestores where either the current path does not exist or it is described by s.
- *Directories.* Specs `dir` $\{\overline{x\ \texttt{is}\ s;}\}$ describe filestores whose contents are described by the sub-specifications s_1 to s_n. A sub-specification s_j may refer to the representation and metadata for s_i using x_i, provided i is less than j.
- *Predicates.* Specs s `where` e describe filestores that are described by s and where e evaluates to $true$.
- *Comprehensions.* Specs $[s \mid x \in e]$ describe filestores with a collection of contents described by s, where x ranges over each element of the collection denoted by e. This generalizes straightforwardly to multiple variables.

The most important artifacts generated from a Forest specification are the `load` function, the `store` function, and the types of the in-memory representation and metadata. Figure 2 (b) gives the representation and metadata types for each specification. The `load` function takes a path as an argument and loads data from the file system, returning a representation and metadata. The `store` function takes both a path and a representation-metadata pair and writes them back to the file system at the given path. These functions enjoy natural "round-tripping" properties—e.g., storing and loading again returns the same representation and metadata. See the Classic Forest paper for formal definitions of the `load` and `store` functions and proofs of these properties [4].

4. 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 forces 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 the running example from Section 2 and suppose that we decide to add a delay to the comprehension:

```
d2 = dir {
  index is "index.txt" :: file;
  data is <[ "data" :: f :: file
           | f <- $lines index$ ]> }
```

If we invoke `d2_load`, we get a representation:

```
{ index = "a.txt\nb.txt"; data = () }
```

Then, to obtain the contents of the files in the `data` directory, 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 Forest 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 Forest run-time system defines

$x \in \mathit{Var}$ *Variables*
$e \in \mathit{Exp}$ *Expressions*
$s ::=$ `file` *Files*
| `link` *Links*
| $e :: s$ *Paths*
| s `option` *Options*
| `dir` $\{\overline{x \texttt{ is } s;}\}$ *Directories*
| s `where` e *Predicates*
| $[s \mid x \in e]$ *Comprehensions*

(a)

s	$\mathcal{R}[\![s]\!]$	$\mathcal{M}[\![s]\!]$
`file`	`string`	`unit md`
`link`	`filepath`	`unit md`
$e :: s$	$\mathcal{R}[\![s]\!]$	(`filepath` $\times \mathcal{M}[\![s]\!]$) `md`
s `option`	$\mathcal{R}[\![s]\!]$ `option`	($\mathcal{M}[\![s]\!]$ `option`) `md`
`dir` $\{\overline{x \texttt{ is } s;}\}$	$\{\overline{x : \mathcal{R}[\![s]\!];}\}$	$\{\overline{x : \mathcal{M}[\![s]\!];}\}$ `md`
s `where` e	$\mathcal{R}[\![s]\!]$	(`boolean` $\times \mathcal{M}[\![s]\!]$) `md`
$[s \mid x \in e]$	$\mathcal{R}[\![s]\!]$ `list`	($\mathcal{M}[\![s]\!]$ `list`) `md`

(b)

Figure 2. Classic Forest: (a) syntax; (b) representation and metadata types.

$s ::= \ldots$
| $\langle s \rangle$ *Delay*

(a)

s	$\mathcal{R}[\![s]\!]$	$\mathcal{M}[\![s]\!]$
$\langle s \rangle$	($\mathcal{R}[\![s]\!] \times \mathcal{M}[\![s]\!]$) `cursor`	`unit md`

(b)

Figure 3. iForest: (a) syntax; (b) representation and metadata types.

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
```

Note that unlike the specification-specific `load` and `store` functions generated by the Classic 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 (d2_new "/")
```

we now get a representation

```
{ index = "a.txt\nb.txt"; data = cur }
```

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

```
load cur
```

to yield the representation:

```
["Hello"; "World!"]
```

Cursors encapsulate the run-time details related to incremental navigation of a filestore, which greatly simplifies applications written 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 = d2_new "/" in
let rep1,md1,n1 = load cur in
let rep2,md2,n2 = load r1.data 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
```

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:

```
d2_new "/" >>= fun cur ->
load cur >>= fun (rep1,md1) ->
load rep1.data >>= fun (rep2,md2) ->
...
```

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. In the future, we plan to explore using `CursorM` as the basis for building a transactional version of iForest, in the style of Haskell's Software Transactional Memory [8].

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

```
d3 = dir {
  index is <"index.txt" :: file>;
  data is <[ "data" :: f :: file
           | f <- $lines index$ ]> }
```

As written, this program will not compile because `index` is a cursor, not a string, which is the type expected by the `lines` function. To fix this error, the programmer would have to explicitly `load` the cursor and apply lines to the resulting string. However, we think 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 6). 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:

```
d4 = dir {
  index is <"index.txt" :: file>;
  data is [ "data" :: f :: file
          | f <- $lines index$ ] }
```

Here, the delayed `index` 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 paper.

- 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. We plan to design a mechanism to check `store` dependencies in the future.

s	$\mathcal{C}[\![s]\!]\ \pi$
`file`	$\mathbf{c}_{\texttt{file}}(v)$ where $v = (\!\vert\texttt{file}\vert\!)\ \pi$
`link`	$\mathbf{c}_{\texttt{link}}(v)$ where $v = (\!\vert\texttt{link}\vert\!)\ \pi$
$e :: s$	$\mathcal{C}[\![s]\!]\ (\pi @ [\![e]\!])$
s `option`	$\begin{cases}\mathbf{0} & \text{if None} = (\!\vert s\ \texttt{option}\vert\!)\ \pi \\ \mathcal{C}[\![s]\!] & \text{otherwise}\end{cases}$
`dir` $\{\overline{x\ \texttt{is}\ s;}\}$	$\mathcal{C}[\![s_1(\rho_1)]\!]\ \pi \cdot \ldots \cdot \mathcal{C}[\![s_n(\rho_n)]\!]\ \pi \cdot \mathbf{c}_{\texttt{dir}}(\overline{x})$ where $\{\overline{x = v}\} = (\!\vert\texttt{dir}\ \{\overline{x\ \texttt{is}\ s;}\}\vert\!)\ \pi$ and $\rho_i = [v_1/x_1, \ldots, v_{i-1}/x_{i-1}]$
s `where` e	$\mathcal{C}[\![s]\!]\ \pi$
$[s \mid x \in e]$	$\mathcal{C}[\![s[v_i/x]]\!]\ \pi \cdot \ldots \cdot \mathcal{C}[\![s[v_k/x]]\!]\ \pi$ where $[v_1, \ldots, v_k] = [\![e]\!]$
$\langle s \rangle$	$\mathbf{0}$

Figure 4. iForest cost model.

5. Cost Model

iForest is designed to enable programmers to make precise tradeoffs between simplicity and performance in applications that store persistent data using the file system. 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, $\mathbf{c}_\tau$, one for each primitive τ (i.e., files, links, and directories). We let π range over file system paths and let @ denote the concatenation operator on paths. We write $v = (\!|s|\!)\ \pi$ if loading with s at π yields v. Similarly, we write $v = [\![e]\!]$ if evaluating e yields v. We write $s[v/x]$ for the substitution of v for x in s. We let ρ range over substitutions and write $s(\rho)$ for the application of ρ to s.

Figure 4 presents the formal definition of the cost model, using the additional notation just defined. The cost $\mathcal{C}[\![s]\!]$ for each specification s is defined with respect to a path π. The cost for a file or link specification (`file` or `link`) 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 π according to e. The cost for an option specification (s `option`) is $\mathbf{0}$ if the file system does not have a node at π and the cost for s otherwise. The cost for a directory is obtained by combining the costs for each field in the directory, as well as the primitive cost function for directories using the monoid operation $(\cdot)$. However, additional care is needed to handle data dependencies: for each field x_i with representation v_i, we substitute v_i for x_i in all subsequent fields. The cost for a predicate specification (s `where` e) is simply the cost for s. Finally the cost for a comprehension is obtained by combining the costs for s with v_i substituted for x, for each i from 1 to k.

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 1 (Delay Monotonicity)**.** *Let $\mathbb{C} = (C, \mathbf{0}, \cdot, \sqsubseteq)$ 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:*

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

The reason for the requirement ($x \cdot z \sqsubseteq x \cdot y \cdot z$) is that any sub-specification may be delayed in a `dir` 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. iForest'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 $\mathbf{c}_\tau\ f$ is 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.

$h ::= \langle\ \rangle$	$s \in \mathbf{Spec}$
$\mid\ \ \rangle\ \langle$	$h \in \mathbf{Skin}$
$\mid\ \sim$	$e \in \mathbf{Expr}$
$\mid\ \_$	$n \in \mathbf{Fields}$
$\mid\ h\ option$	$\Phi \in (\mathbf{DTree} \to \mathbb{B})$
$\mid\ \{\overline{h}\}$	
$\mid\ [h]$	
$\mid\ h \vert \Phi$	
$\mid\ h_1 + h_2$	$[\![\cdot]\!]_h : \mathbf{Skin} \to \mathbf{DTree} \rightharpoonup \mathbf{DTree}$
$\mid\ h_1; h_2$	$dtreeof : \mathbf{Spec} \to \mathbf{DTree}$
$\mid\ map(h)$	$apply : \mathbf{Spec} \to \mathbf{DTree} \rightharpoonup \mathbf{Spec}$
$\mid\ n(h)$	

Figure 5. Skin language syntax.

- $\mathbb{C} = (\mathbb{R}_+, 0, +)$, $\sqsubseteq$ is $\leq$ on the real numbers and $\mathbf{c}_\tau\ f$ is the amount of time it took to load the file or link and 0 for directories (since loading all 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 $\mathbf{c}_{\mathbf{file}}\ f = 1$ and $\mathbf{c}_{\mathbf{link}}\ f = \mathbf{c}_{\mathbf{dir}}\ f = 0$: The total cost of a specification will be the number of files (not including links) loaded.
- $\mathbb{C} = (\mathbb{M}, \emptyset, \cup)$ where $\mathbb{M}$ is a multiset of files, $\sqsubseteq$ is the subset relation on multisets and $\mathbf{c}_\tau\ f$ is the name of the file, link, or directory: The total cost of a specification will be the multiset containing the names of everything loaded.

Note that all of these examples have the monotonicity property of Theorem 1. As discussed in Section 4, 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.

6. Skins

iForest's 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 would have three options, all of which would be unattractive:

1. They could only delay 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's skins provides 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 they can be applied to. 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 C.

Figure 5 defines the syntax of the skin language and gives the types of 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 ($h\ option$), directory ($\{\overline{h}\}$), and comprehension ($[h]$) skins modify their sub-specifications. The predicate skin ($h|\Phi$) applies h only if Φ is satisfied. The union skin ($h_1 + h_2$) applies h_1 if possible and otherwise applies h_2. The composite skin ($h_1; h_2$) applies h_1 and h_2 in sequence. The map skin ($map(h)$) applies h to each sub-specification. Finally, the match skin ($n(h)$) applies h to a directory field specified by n.

To a first approximation, a skin can be thought of as denoting a tree 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 ($dtreeof$) and we can apply a delay tree to a specification to obtain a new specification ($apply$).

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

```
d = dir {
  index is "index.txt" :: file;
  data is [ "data" :: f :: file
          | f <- $lines index$ ] }
```

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

```
d1Skin = {_,[<>]}
d2Skin = data(<>)
d3Skin = {<>,<>}
d4Skin = {~,_}
```

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

```
d1 = d @ d1Skin
d2 = d @ d2Skin
d3 = d @ d3Skin
d4 = d @ d4Skin
```

Skin `d1Skin` modifies `d` by first applying the identity skin (_) to the first part of the directory, then applying the delay skin ($\langle\ \rangle$) inside the comprehension of the second part of the specification, generating:

```
d1 = dir {
  index is "index.txt" :: file;
  data is [ "data" :: f :: <file>
          | f <- $lines index$ ] }
```

Note that there is no distinction between delaying the whole path construct, `"data" :: f :: file`, and just `file`. `d2Skin` modifies `d` by matching on `data` and then applying the delay skin. This matching operation can be done on any named field in a directory. The result is:

```
d2 = dir {
  index is "index.txt" :: file;
  data is <[ "data" :: f :: file
          | f <- $lines index$ ]> }
```

Skin `d3Skin` simply delays both parts of the directory:

```
d3 = dir {
  index is <"index.txt" :: file>;
  data is <[ "data" :: f :: file
          | f <- $lines index$ ]> }
```

Skin `d4Skin` flips the delay annotation on the first part of the directory (i.e. the index), which becomes delayed since it was not previously, and we end up with:

```
d4 = dir {
  index is <"index.txt" :: file>;
  data is [ "data" :: f :: file
          | f <- $lines index$ ] }
```

Finally, consider a new set of delay annotations on `d`:

```
d5 = dir {
  index is "index.txt" :: <file>;
  data is [ "data" :: f :: <file>
          | f <- $lines index$ ] }
```

There is a useful skin idiom we can use to achieve this result. The idiom delays everything with a particular type, usually 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 Φ. In this case, we use the built-in predicate `file`, which tests whether the type of the specification is a file. Appendix C shows the collection of built-in predicates. Skin `delayFiles` 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, $map(h)$ desugars into $[h] + h\ option + \{h, .., h\} + \_$. Skin `delayFiles` maps itself, which means it will apply itself to every sub-specification (or do nothing if it is at a leaf node). If we apply `delayFiles` to `d`, we get `d5`.

Both the `delayAll` skin (shown in Section 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 Φ 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 D shows eight theorems and nine 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 C.

7. Experience

We built prototype implementations of iForest and skins, as well as a version of Pads [2], 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 [13] (Figure 7 (a)), and we built several applications in collaboration with the Cornell Soil and Water Lab, which is led by Todd Walter. We conducted experiments showing skins can speed up load times by approximately 7x.

SWAT Overview. The Soil and Water Assessment Tool (SWAT) [13] 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, for example

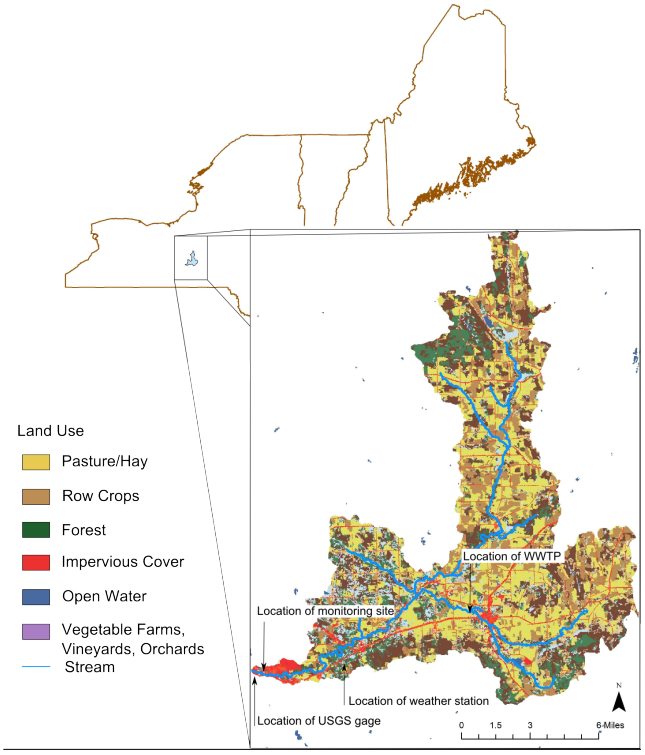


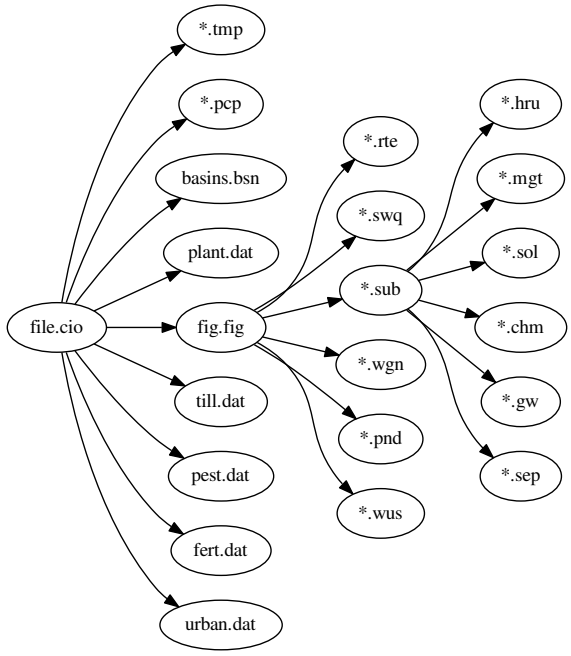
Figure 6. The Fall Creek watershed

by replacing a forest with a housing development. In an initialization 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 can not be spread over multiple subbasins. In our examples, we used a SWAT initialization from the Fall Creek watershed in Ithaca, NY, pictured in Figure 6.

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 examples), 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 7(b) 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.

```
swatIn = dir {
  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 <- $allWusFiles subs$ ] }
```

(a)



(b)

Figure 7. SWAT filestore: specification and dependencies. The constants in the specification that are not `file` are Pads specifications describing the contents of the individual files.

Example Application: Calibration. An important first step in any SWAT application is to calibrate the model to ensure 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 [11]. 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.

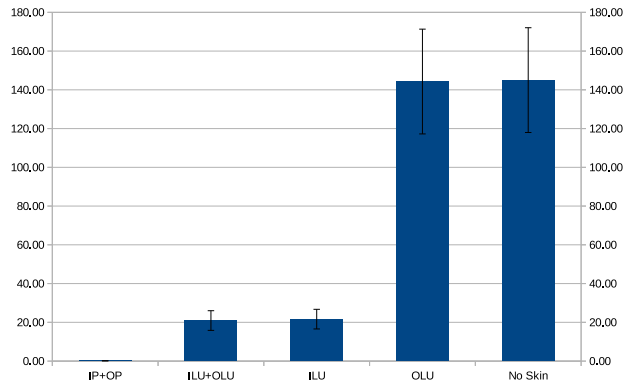
Example Application: Management. After calibrating, many applications can be built using SWAT. One common use is quantifying the impact of various land management decisions on a watershed [5–7, 12, 16]. This involves encoding management decisions as inputs to the model and then interpreting model output. Operationally, this application is similar to calibration in that the scientist modifies input parameters stored 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 7(a) 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`.

iForest 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 they are interested in while sharing a single iForest description across many different applications.

Results in Brief. While writing an initial specification can be time consuming because of the myriad details (see the SWAT manual[14]), once we had the specification, writing applications using it was generally straightforward. We found the skin language expressive enough to describe everything necessary in a few lines. 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



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 8. Load times for SWAT input-output directories.

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.

7.1 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 8). We used the specifications in Figure 9 with 5 different levels of skinning 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; `swatOut` is an entirely undelayed specification of the seven output files of a SWAT execution.
- OLU, which loaded the `swatIn` and `swatOLU` specifications respectively. `swatIn` is entirely undelayed.
- NoSkin, which loaded the `swatIn` and `swatOut` specifications.

Figure 8 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.

7.2 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.

```
(* 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 9. SWAT Skins and the resulting iForest specifications. The `swatIn` specification appears in Figure 7; `swatOut` is not shown.

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 [11],

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

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 to simply predict the averge of the observed data at every point. Generally, $E > 0.5$ is considered satisfactory [10].

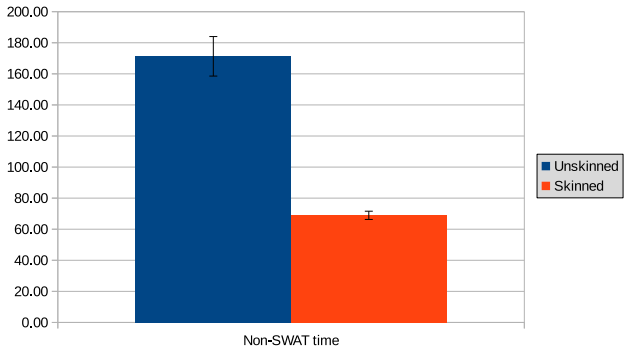
Figure 10 lists a set of parameters that are relevant for calibrating the Fall Creek watershed, showing that the search space is extremely large. In our application, we only modified 4 parameters, chosen because they are especially sensitive: ALPHA_BF, GW_DELAY, SURLAG, and ESCO. We picked 6 points per parameter, distributed relatively evenly over the search space, and ran calibration. Specifically, we wrote all combinations 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* that the value of 0.621 that our colleagues had previously obtained.

Figure 11 gives the running time of our calibration application. The majority of the time comes from running SWAT, which takes ˜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-

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 10. Calibration parameters.



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

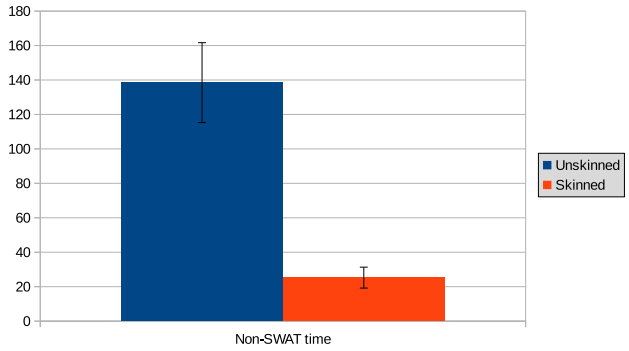
Figure 11. Calibration experiment.

SWAT time, presumably because the program is no longer just loading. Even so, the skinned version is notably faster.

7.3 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.* [5–7, 12, 16]. 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



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 12. Land management experiment.

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 12 and 13 show our results. The first figure reports timing information, 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.

8. Related Work

The work in this paper builds upon earlier work on the design of Forest [4]. iForest differs from Forest in the introduction of delays and skins. The original version of Forest was implemented in Haskell and uses Haskell's inherent laziness to try to avoid loading unnecessary portions of the filestore. This approach required using unsafe extensions of the language, however, because file system manipulation has inherent side effects—i.e., in the I/O monad. In contrast, iForest is implemented in OCaml and requires users to explicitly manage loading of filestore components using delays and skins.

iForest leverages work from the Pads project [2, 3]. Pads uses data-dependent type declarations to describe the structure and invariants of data in a single file. From such specifications, the Pads compiler generates types for parsed data and a suite of data-processing tools. The most significant difference between Pads (and other parser generators) and Forest is that Pads generates infrastructure to process individual files whereas iForest generates infrastructure to process entire filestores. Pads does not support incremental loading.

As with Forest, iForest shares the goal of systems like Microsoft's LINQ [9] and F#'s Type Providers [15] of making data-oriented programming easier. iForest differs from both LINQ and Type Providers in that neither of those systems support the declarative specification of filestores. XML-based languages like XFiles [1] do allow the declarative

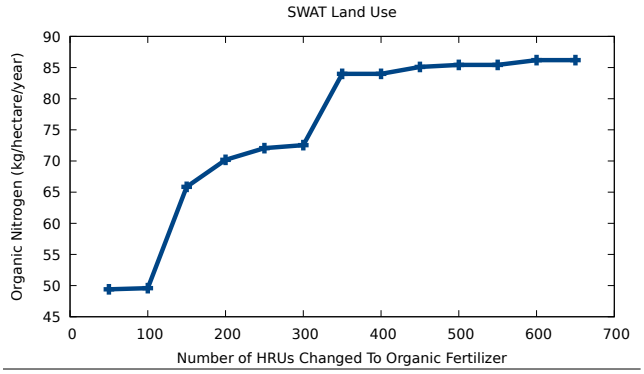


Figure 13. Increase in organic nitrogen as more HRUs use Fresh Dairy Manure fertilizer.

specification of filestores, but they are not tightly integrated into a host language, and so they do not provide the easy access to filestore data that we seek to provide.

9. Conclusion

We presented Incremental Forest, a domain-specific language that extends Forest [4] with a new *delay* construct to enable processing file system data incrementally. 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 skin language, which allows programmers to induce different delay annotations in specifications without rewriting them. We described a type system to ensure 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 management application 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.

Acknowledgments

The authors wish to thank the Cornell PLDG, Don Syme, David Walker, and the anonymous OOPSLA reviewers for helpful comments on an earlier draft of this paper. Our work is supported in part by the National Science Foundation under grants CCF-1253165 and CNS-CNS-1413972, and gifts from Cisco, Facebook, Fujitsu, and Google.

References

[1] S.-C. Buraga. An XML-based semantic description of distributed file systems. In *RoEduNet*, 2003.

[2] K. Fisher and R. Gruber. PADS: A domain specific language for processing ad hoc data. In *PLDI*, June 2005.

[3] K. Fisher, Y. Mandelbaum, and D. Walker. The next 700 data description languages. *J. ACM*, 57, February 2010.

[4] K. Fisher, N. Foster, D. Walker, and K. Q. Zhu. Forest: A language and toolkit for programming with filestores. In *ICFP*, 2011.

[5] M. Gabriel, C. Knightes, E. Cooter, and R. Dennis. 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.

[6] K. K. Garg, L. Bharati, A. Gaur, B. George, S. Acharya, K. Jella, and B. Narasimhan. Spatial mapping of agricultural water productivity using the SWAT model in Upper Bhima Catchment, India. *Irrigation and Drainage*, 61(1), 2012.

[7] B. Guse, M. Pfannerstill, and N. Fohrer. Dynamic modelling of land use change impacts on nitrate loads in rivers. *Environmental Processes*, 2(4), 2015.

[8] T. Harris, S. Marlow, S. Peyton Jones, and M. Herlihy. Composable memory transactions. In *PPoPP*, 2005.

[9] LINQ: .NET Language-Integrated Query. LINQ: .NET language-integrated query. `http://msdn.microsoft.com/library/bb308959.aspx`, Feb. 2007.

[10] D. N. Moriasi, J. G. Arnold, M. W. Van Liew, R. L. Bingner, R. D. Harmel, and T. L. Veith. Model evaluation guidelines for systematic quantification of accuracy in watershed simulations. *Transactions of the ASABE*, 50(3), 2007.

[11] J. Nash and J. Sutcliffe. River flow forecasting through conceptual models part I — A discussion of principles. *Journal of Hydrology*, 10(3), 1970.

[12] T. S. Ngo, D. B. Nguyen, and P. S. Rajendra. 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.

[13] SWAT. Soil and Water Assessment Tool. `http://swat.tamu.edu`.

[14] SWAT IO Documentation. SWAT Input/Output Documentation. `http://swat.tamu.edu/documentation/2012-io/`, 2012.

[15] D. Syme. Looking ahead with F#: Taming the data deluge. Presentation at the Workshop on F# in Education, Nov. 2010.

[16] C. B. Zou, L. Qiao, and B. P. Wilcox. Woodland expansion in central Oklahoma will significantly reduce streamflows – a modelling analysis. *Ecohydrology*, 2015.

A. 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 [4] with a delay construct. Its syntax is as follows:

$$
\begin{array}{lll}
\textit{Paths } r & ::= & \cdot \mid r \,/\, u \\
\textit{Contents } T & ::= & \mathsf{File}\ u \mid \mathsf{Link}\ r \\
\textit{File systems } F & ::= & \{\!|\, r_1 \mapsto (a_1, T_1), .., r_n \mapsto (a_k, T_n) \,|\!\} \\
\textit{Specifications } s & ::= & k_{\tau_1}^{\tau_2} \mid e :: s \mid s? \mid (x{:}s_1, s_2) \\
 & & \mid P(e) \mid [s \mid x \in e] \mid \mathit{Delay}(s)
\end{array}
$$

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:

- `file` and `link` are translated to constants $k^{\tau_2}_{\tau_1}$, where types τ_1 and τ_2 are appropriate representation and metadata types.
- s `option` is translated to $s?$
- `dir` $\{\overline{x \texttt{ is } s;}\}$ is translated to $(\_{:}\mathsf{dir}^{unit}_{unit}, (x_1{:}s_1, (x_2{:}s_2, (\ldots (x_{n-1}{:}s_{n-1}, s_n))))$, 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 $(this{:}s, P(e))$. 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.

B. iForest Semantics

This section defines the formal semantics of iForest and proves several round-tripping properties.

Figure 14 (a) gives the representation and metadata types for each specification. The type τ `md` denotes the pair $(\texttt{bool} * \tau)$, where the boolean value indicates if an error occurred during loading. We write τ `cur` for the type of a cursor that returns τ when forced. Figure 14 (b-c) 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 [4]. We explain the new rules in detail.

Loading. The judgment $\mathcal{E} \vdash \texttt{load}\,(F, r, s) \triangleright (v, md)$ holds if, in environment $\mathcal{E}$, when we load file system F at path r into memory as a specification s, we get a pair with representation v and metadata md. We use a number of auxiliary functions. First, we use functions $load_k(\mathcal{E}, F, r)$ to implement the load function for each constant $k^{\tau_2}_{\tau_1}$. These functions take an environment $\mathcal{E}$, a file system F, and a path r as arguments and return the representation and metadata of type τ_1 and τ_2 respectively. Second, the operator $[\![e]\!]^{\mathcal{E}}_{\tau}$ evaluates expression e in environment $\mathcal{E}$ and returns a value of type τ. Third, we use standard projection functions π_1 and π_2.

The first new rule introduced in iForest is for loading a delayed specification. This rule returns as its representation a cursor (denoted $\mathbf{c}_{(\mathcal{E},r,s)}$) 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 $(true, ())$.

The second new rule uses a different judgment of the form $\mathcal{E}' \vdash \texttt{load}_{\mathbf{c}}(\mathbf{c}_{(\mathcal{E},r,s)}, F) \triangleright (v, md)$. This judgment holds if, in environment $\mathcal{E}'$, when we load cursor $\mathbf{c}_{(\mathcal{E},r,s)}$ in file system F, we get representation and metadata pair (v, md). The rule says that if we perform a normal load in the given file system, F, with the environment, path, and specification encapsulated in the cursor, we will get the same result as when we perform a cursor-load in F in any environment.

Storing. The judgment $\mathcal{E} \vdash \texttt{store}\,(F, v, md, r, s) \triangleright (F', \varphi')$ holds if, in environment $\mathcal{E}$ and file system F, storing representation v and metadata md at path r using specification s produces a new file system F' and validator φ'.

A validator is a predicate on file systems that checks for internal inconsistencies in the representation and metadata [4]. We say that storing passes validation if the validator returns $true$ when evaluated on the resulting filesystem. As with the load function, we use several auxiliary functions to define the store function. First, we use functions $store_k(\mathcal{E}, F, r, v, md)$ to implement the store function for each constant $k^{\tau_2}_{\tau_1}$. This function takes as arguments an environment $\mathcal{E}$, a file system F, a path r, a representation v and metadata md. Second, we use an append operation on file systems, F_1++F_2. Intuitively, this operation copies all contents from F_2 to F_1, overwriting any contents they have in common. Third, the $F[r := \bot]$ function removes the mapping for a path r in F, or returns F if $r \notin dom(F)$. Finally, the function $md^s_{default}$ computes "default" metadata for s.

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 $\mathcal{E}' \vdash \texttt{store}_{\mathbf{c}}(\mathbf{c}_{(\mathcal{E},r,s)}, F, v, md) \triangleright (F', \varphi')$. This holds if, in environment $\mathcal{E}'$, storing cursor $\mathbf{c}_{(\mathcal{E},r,s)}$ into file system F with representation v and metadata md yields a new file system and validator pair (F', φ'). 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 2 (LoadStore)**.** *Let $\mathcal{E}$ be an environment, F and F' file systems, r a path, s a specification, v a representation, md metadata, and φ' a validator. If*

$$\mathcal{E} \vdash \texttt{load}\,(F, r, s) \triangleright (v, md)$$
$$\mathcal{E} \vdash \texttt{store}\,(F, v, md, r, s) \triangleright (F', \varphi')$$

then $F = F'$ and $\varphi'(F')$.

s	$\mathcal{R}[\![s]\!]$	$\mathcal{M}[\![s]\!]$
$k^{\tau_2}_{\tau_1}$	τ_1	τ_2 `md`
$e :: s$	$\mathcal{R}[\![s]\!]$	$\mathcal{M}[\![s]\!]$
$(x{:}s_1, s_2)$	$\mathcal{R}[\![s_1]\!] * \mathcal{R}[\![s_2]\!]$	$(\mathcal{M}[\![s_1]\!] * \mathcal{M}[\![s_2]\!])$ `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`

(a)

$$\frac{}{\mathcal{E} \vdash \texttt{load}\ (F, r, k^{\tau_2}_{\tau_1}) \rhd load_k(\mathcal{E}, F, r)}$$

$$\frac{\mathcal{E} \vdash \texttt{load}\ (F, [\![r/e]\!]^{\mathcal{E}}_{\texttt{filepath}}, s) \rhd (v, md)}{\mathcal{E} \vdash \texttt{load}\ (F, r, e :: s) \rhd (v, md)}$$

$$\frac{\begin{array}{c}\mathcal{E} \vdash \texttt{load}\ (F, r, s_1) \rhd (v_1, md_1) \\ (\mathcal{E}, x \mapsto v_1, x_{md} \mapsto md_1) \vdash \texttt{load}\ (F, r, s_2) \rhd (v_2, md_2) \\ b = ((\pi_1\ md_1) \wedge (\pi_1\ md_2))\end{array}}{\mathcal{E} \vdash \texttt{load}\ (F, r, (x{:}s_1, s_2)) \rhd ((v_1, v_2), (b, (md_1, md_2)))}$$

$$\frac{\begin{array}{c}[\![e]\!]^{\mathcal{E}}_{\alpha\ \texttt{list}} = [w_1, .., w_k] \\ \forall i \in \{1, .., k\}.(\mathcal{E}, x \mapsto w_i) \vdash \texttt{load}\ (F, r, s) \rhd (v_i, md_i) \\ b = \bigwedge^k_i \pi_1\ md_i \wedge vs = [v_1, .., v_k] \wedge mds = [md_1, .., md_k]\end{array}}{\mathcal{E} \vdash \texttt{load}\ (F, r, [s \mid x \in e]) \rhd (vs, (b, mds))}$$

$$\frac{b = [\![e]\!]^{\mathcal{E}}_{\texttt{bool}}}{\mathcal{E} \vdash \texttt{load}\ (F, r, P(e)) \rhd ((), (b, ()))}$$

$$\frac{r \in dom(F) \wedge \mathcal{E} \vdash \texttt{load}\ (F, r, s) \rhd (v, md)}{\mathcal{E} \vdash \texttt{load}\ (F, r, s?) \rhd (v, md)}$$

$$\frac{r \notin dom(F)}{\mathcal{E} \vdash \texttt{load}\ (F, r, s?) \rhd (None, (true, None))}$$

$$\frac{}{\mathcal{E} \vdash \texttt{load}\ (F, r, \langle s \rangle) \rhd (\mathbf{c}_{(\mathcal{E}, r, s)}, (true, ()))}$$

$$\frac{\mathcal{E} \vdash \texttt{load}\ (F, r, s) \rhd (v, md)}{\mathcal{E}' \vdash \texttt{load}_{\mathbf{c}}(\mathbf{c}_{(\mathcal{E}, r, s)}, F) \rhd (v, md)}$$

(b)

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

$$\frac{\mathcal{E} \vdash \texttt{store}\ (F, v, md, [\![r/e]\!]^{\mathcal{E}}_{\texttt{filepath}}, s) \rhd (F', \varphi')}{\mathcal{E} \vdash \texttt{store}\ (F, v, md, r, e :: s) \rhd (F', \varphi')}$$

$$\frac{\begin{array}{c}md = (b, (md_1, md_2)) \wedge v = (v_1, v_2) \\ \mathcal{E}' = (\mathcal{E}, x \mapsto v_1, x_{md} \mapsto md_1) \\ b' = (b = (\pi_1\ md_1) \wedge (\pi_1\ md_2)) \\ \mathcal{E} \vdash \texttt{store}\ (F, v_1, md_1, r, s_1) \rhd (F_1, \varphi_1) \\ \mathcal{E}' \vdash \texttt{store}\ (F, v_2, md_2, r, s_2) \rhd (F_2, \varphi_2) \\ \varphi' = (\lambda F'.b' \wedge\ \varphi_1(F') \wedge \varphi_2(F'))\end{array}}{\mathcal{E} \vdash \texttt{store}\ (F, v, md, r, (x{:}s_1, s_2)) \rhd (F_1 \texttt{++} F_2, \varphi')}$$

$$\frac{\begin{array}{c}vs = [v_1, .., v_j] \wedge mds = [md_1, .., md_l] \\ [\![e]\!]^{\mathcal{E}}_{\alpha\ \texttt{list}} = [w_1, .., w_m] \wedge k = min(j, l, m) \\ b' = (b = \bigwedge^k_i \pi_1\ md_i) \wedge \forall i \in \{1, .., k\}. \\ (\mathcal{E}, x \mapsto w_i) \vdash \texttt{store}\ (F, v_i, md_i, r, s) \rhd (F_i, \varphi_i) \\ \varphi' = (\lambda F'.(j = l = m) \wedge b' \wedge (\bigwedge^k_i \varphi_i(F'))) \\ F' = F_1 \texttt{++} .. \texttt{++} F_k\end{array}}{\mathcal{E} \vdash \texttt{store}\ (F, vs, (b, mds), r, [s \mid x \in e]) \rhd (F', \varphi')}$$

$$\frac{\varphi' = \lambda F'.b = [\![e]\!]^{\mathcal{E}}_{\texttt{bool}}}{\mathcal{E} \vdash \texttt{store}\ (F, (), (b, ()), r, P(e)) \rhd (F, \varphi')}$$

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

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

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

$$\frac{}{\mathcal{E} \vdash \texttt{store}\ (F, v, md, r, \langle s \rangle) \rhd (F, \lambda F'.true)}$$

$$\frac{\mathcal{E} \vdash \texttt{store}\ (F, v, md, r, s) \rhd (F', \varphi')}{\mathcal{E}' \vdash \texttt{store}_{\mathbf{c}}(\mathbf{c}_{(\mathcal{E}, r, s)}, F, v, md) \rhd (F', \varphi')}$$

(c)

Figure 14. iForest semantics: (a) representation and metadata types; (b) load function; (c) store function.

Theorem 3 (StoreLoad)**.** *Let $\mathcal{E}$ be an environment, F and F' file systems, r a path, s a specification, v and v' representations, md and md' metadata, and φ' a validator. If*

$$\mathcal{E} \vdash \texttt{store}\,(F, v, md, r, s) \rhd (F', \varphi') \quad \varphi'(F')$$
$$\mathcal{E} \vdash \texttt{load}\,(F', r, s) \rhd (v', md')$$

then $(v', md') = (v, md)$.

Theorem 4 (iLoadStore)**.** *Let $\mathcal{E}$, $\mathcal{E}'$, and $\mathcal{E}''$ be environments, F and F' file systems, v a representation, md a metadata, φ' a validator, and $\mathbf{c}_{(\mathcal{E},r,s)}$ a cursor. If*

$$\mathcal{E}' \vdash \texttt{load}_{\mathbf{c}}(\mathbf{c}_{(\mathcal{E},r,s)}, F) \rhd (v, md)$$
$$\mathcal{E}'' \vdash \texttt{store}_{\mathbf{c}}(\mathbf{c}_{(\mathcal{E},r,s)}, F, v, md) \rhd (F', \varphi')$$

then $F = F'$ and $\varphi'(F')$.

Theorem 5 (iStoreLoad)**.** *Let $\mathcal{E}$, $\mathcal{E}'$, and $\mathcal{E}''$ be environments, F and F' file systems, v and v' representations, md and md' metadata, φ' a validator, and $\mathbf{c}_{(\mathcal{E},r,s)}$ a cursor. If*

$$\mathcal{E}' \vdash \texttt{store}_{\mathbf{c}}(\mathbf{c}_{(\mathcal{E},r,s)}, F, v, md) \rhd (F', \varphi') \quad \varphi'(F')$$
$$\mathcal{E}'' \vdash \texttt{load}_{\mathbf{c}}(\mathbf{c}_{(\mathcal{E},r,s)}, F') \rhd (v', md')$$

then $(v', md') = (v, md)$.

Note these judgments do not require the same environments since the environment in the cursor is used instead.

C. Skin Core Syntax and Semantics

This section describes the formal syntax and semantics of skins and their accompanying type system.

Syntax. Figure 15 defines the semantics for a skin core calculus. This syntax is a subset of the syntax from Figure 5. For example, instead of having $\{\overline{h}\}$, we have pairs, and we do not have $\rangle\,\langle$, $map(h)$, or $n(h)$, 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 Figure 16 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 $s@h$.

Function $dtreeof$ strips away extraneous information from its argument to generate the corresponding delay tree. 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. 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 ($h\ option$), 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., $[t]$, $t\ option$, 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 the specification to belong to both the specified type and the type of its sub-skin.

Function ($[\![\cdot]\!]_t$) takes a type and a delay tree and checks whether the delay tree belongs to that type. 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. Function $apply$ 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 D), partiality is not an issue in practice.

D. Skin Properties and Theorems

This section presents the main lemmas and theorems 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 1 (Skin Application)**.**

$$s@h \;=\; apply\ s\ ([\![h]\!]_h\ (dtreeof\ s))$$

Metavar Conventions

$s \in \mathbf{Spec}$
$h \in \mathbf{Skin}$
$d \in \mathbf{DTree}$
$t \in \mathbf{Type}$
$x \in \mathbf{Var}$
$e \in \mathbf{Expr}$
$b \in \mathbb{B}$

Function Types

$dtreeof : \mathbf{Spec} \to \mathbf{DTree}$
$typeofD : \mathbf{DTree} \to \mathbf{Type}$
$typeofH : \mathbf{Skin} \to \mathbf{Type}$
$[\![\cdot]\!]_t : \mathbf{Type} \to \mathbf{DTree} \to \mathbb{B}$
$[\![\cdot]\!]_h : \mathbf{Skin} \to \mathbf{DTree} \rightharpoonup \mathbf{DTree}$
$apply : \mathbf{Spec} \to \mathbf{DTree} \rightharpoonup \mathbf{Spec}$

Delay Tree Syntax

$d ::= k^{\tau_2}_{\tau_1}$
$\mid \ (d_1, d_2)$
$\mid \ [d]$
$\mid \ p$
$\mid \ d?$
$\mid \ Delay(d)$

Skin Syntax

$h ::= \langle\,\rangle$
$\mid \ \sim$
$\mid \ \_$
$\mid \ [h]$
$\mid \ h\ option$
$\mid \ \{h_1, h_2\}$
$\mid \ h_1 + h_2$
$\mid \ h_1; h_2$
$\mid \ h|t$

Type Syntax

$t ::= cons^{\tau_2}_{\tau_1}$
$\mid \ p$
$\mid \ [t]$
$\mid \ t\ option$
$\mid \ \{t_1, t_2\}$
$\mid \ t_1 \wedge t_2$
$\mid \ t_1 \vee t_2$
$\mid \ \top$
$\mid \ \bot$

Figure 15. Formal syntax of all components of the skin language

Next, we define skin equivalence:

Definition 2 (Skin Equivalence)**.**

$$h_1 = h_2 \iff ((\![h_1]\!]_h\ d_0 = d \wedge\ [\![h_2]\!]_h\ d_0 = d') \implies d = d')$$

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:

Theorem 6 (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 6 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 7). First we show that application preserves delay trees:

Lemma 1 (Delay Tree Preservation)**.**

$$d \in dom(apply\ s) \implies dtreeof\ (apply\ s\ d) = d$$

Let $=^s_d$ 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 2 (Apply Equivalence)**.**

$$s_1\ =^s_d\ s_2 \implies apply\ s_1\ d = apply\ s_2\ d$$

Lemma 3 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 3 (Apply Preservation)**.**

$$d \in dom(apply\ s) \implies apply\ s\ d\ =^s_d\ s$$

Lemma 4 shows that applying *apply* twice in sequence is equivalent to just the second application.

Lemma 4 (Apply Cancellation)**.**

$$d_1 \in dom(apply\ s) \implies$$
$$apply\ (apply\ s\ d_1)\ d_2 = apply\ s\ d_2$$

Finally, Theorem 7 combines the previous four lemmas to prove that applying two skins to a specification one after another is equivalent to applying their composition.

Theorem 7 (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 8). Lemma 5 shows that the type of a delay tree is not affected by delay annotations.

Lemma 5 (Invariance under Delays)**.**

$$d_1\ =^d_d\ d_2 \implies typeofD\ d_1 = typeofD\ d_2$$

Lemma 6 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 6 (DTreeof Preservation)**.**

$$s_1\ =^s_d\ s_2 \implies dtreeof\ s_1\ =^d_d\ dtreeof\ s_2$$

Theorem 8 shows that the type of a specification is not affected by skin application.

Theorem 8 (Invariance under Skin Application)**.**

$$dtreeof\ s \in dom([\![h]\!]_h) \implies$$
$$typeofD\ (dtreeof\ s) = typeofD\ (dtreeof\ s@h)$$

$$dtreeof : \mathbf{Spec} \to \mathbf{DTree}$$

$$\begin{array}{llll}
dtreeof & k^{\tau_2}_{\tau_1} & = k^{\tau_2}_{\tau_1} \\
dtreeof & e :: s & = dtreeof\ s \\
dtreeof & (x{:}s_1, s_2) & = (dtreeof\ s_1, dtreeof\ s_2) \\
dtreeof & [s \mid x \in e] & = [dtreeof\ s] \\
dtreeof & P(e) & = p \\
dtreeof & s? & = (dtreeof\ s)? \\
dtreeof & Delay(s) & = Delay(dtreeof\ s)
\end{array}$$

$$typeofD : \mathbf{DTree} \to \mathbf{Type}$$

$$\begin{array}{llll}
typeofD & k^{\tau_2}_{\tau_1} & = cons^{\tau_2}_{\tau_1} \\
typeofD & p & = p \\
typeofD & Delay(d) & = typeofD\ d \\
typeofD & [d] & = [typeofD\ d] \\
typeofD & d? & = (typeofD\ d)\ option \\
typeofD & (d_1, d_2) & = \{typeofD\ d_1, typeofD\ d_2\}
\end{array}$$

$$typeofH : \mathbf{Skin} \to \mathbf{Type}$$

$$\begin{array}{llll}
typeofH & \langle\,\rangle & = \top \\
typeofH & \sim & = \top \\
typeofH & _ & = \top \\
typeofH & [h] & = [typeofH\ h] \\
typeofH & h\ option & = (typeofH\ h)\ option \\
typeofH & \{h_1, h_2\} & = \{typeofH\ h_1, typeofH\ h_2\} \\
typeofH & h_1; h_2 & = (typeofH\ h_1) \wedge (typeofH\ h_2) \\
typeofH & h_1 + h_2 & = (typeofH\ h_1) \vee (typeofH\ h_2) \\
typeofH & h|t & = t \wedge (typeofH\ h)
\end{array}$$

$$[\![\cdot]\!]_t : \mathbf{Type} \to \mathbf{DTree} \to \mathbb{B}$$

```
⟦t⟧t d =
  match d with
  | Delay(d) -> ⟦t⟧t d
  | d ->
    match (t,d) with
    | (cons^τ2_τ1, k^τ2_τ1) -> true
    | (p,p) -> true
    | (t option, 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
```

$$[\![\cdot]\!]_h : \mathbf{Skin} \to \mathbf{DTree} \rightharpoonup \mathbf{DTree}$$

```
⟦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 option, 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
```

$$apply : \mathbf{Spec} \to \mathbf{DTree} \rightharpoonup \mathbf{Spec}$$

```
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^τ2_τ1, k^τ2_τ1) -> k^τ2_τ1
    | (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 16. Formal semantics of skins

This property is important because it means that users do not have to consider anything but 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 show that the type system is sound and complete. Lemma 7 shows that type checking only depends on the structure of a delay tree, not the delays.

Lemma 7 (Typing Invariance under Delays)**.**

$$(d_1 =_d^d d_2 \wedge [\![t]\!]_t\, d_1 \implies [\![t]\!]_t\, d_2)$$

Lemma 8 shows that the underlying structure of a delay tree is not changed by skin application.

Lemma 8 (Skin Application Preservation)**.**

$$[\![h]\!]_h\, d_1 = d_2 \implies d_1 =_d^d d_2$$

Lemma 9 shows that only the underlying structure of a delay tree determines whether or not a skin can be applied to it.

Lemma 9 (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 9 (Soundness)**.** *The type system is sound,* i.e.*,*

$$[\![typeofH\ h]\!]_t\, d \implies d \in dom([\![h]\!]_h)$$

Theorem 10 (Completeness)**.** *The type system is complete,* i.e.*,*

$$d \in dom([\![h]\!]_h) \implies [\![typeofH\ h]\!]_t\, d$$