Formally-Based Resource Usage Verification using a Dependently-Typed MetaLanguage to Specify and Implement Domain-Specific Languages

Abstract

In the modern, multi-threaded, multi-core programming environment, correctly managing system resources such as locks or file handles can be especially difficult and error prone. A simple error, such as forgetting to release a lock, can have major consequences on the correct operation of the program (e.g. deadlock), often at a time and location that is isolated from the original error.

While there have been many previous proposals for managing locks and resources, these often arise from the systems arena, and are therefore integrated only loosely with the host programming language. Such approaches therefore do not generally provide the strong formal guarantees of *correctness-by-construction* that we would prefer for all fundamental properties of our programs. In this paper, we propose a new type-based approach to resource management, based on the use of *dependent types* to construct a Domain Specific Language whose typing rules enforce the properties we require. We illustrate our approach with reference to simple file manipulating operations on a Unix-like file system, and also with reference to a concurrent multi-threaded environment.

1. Introduction

As multi-core architectures become more common, so approaches to concurrency based on serialising access to system resources become a major potential performance bottleneck. Truly concurrent access to system resources [13], where any thread may initiate an I/O request in any required order, is essential to provide high-performance in this modern setting. However, locking and other concurrency issues make this a difficult and error-prone environment for the typical programmer. It is our contention that if acceptable safety is to be maintained without sacrificing important potential performance benefits, programmer access to system resources should ideally be through *verified* implementations of formally-specified resource protocols, supported by appropriate programming language abstractions. A good abstraction will both ensure safety and minimise locking and other overheads. In this way, we aim to achieve “correctness-by-construction” [4], where any valid program is guaranteed to possess certain properties. Using such an approach both improves reliability and reduces development costs, through eliminating unnecessary testing and debugging steps.

[copyright notice will appear here]

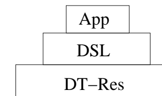

Figure 1. System Overview

Popeea and Chin observe [24] that there are two key facets to verifying resource access protocols: *protocol verification*, to ensure that the protocol conforms to the required formal propeties; and *usage verification*, to ensure that resources are accessed and used in accordance with the protocol. We have previously considered the former problem, exploiting dependent types to yield programs that conform, by construction, to required bounds on resource consumption, in terms of time, memory or other resources [1, 2]. This paper considers the complementary problem of resource usage verification under the assumption of an already verified protocol. We use a type-based approach to managing locks and other system resources, such as file handles, to construct a simple dependently-typed Domain-Specific Language (DSL). The typing rules of this DSL then directly enforce the required safety properties, such as safe acquisition and release of resource locks. Figure 1 gives a pictorial overview of the system; we build a domain specific language (the object language) on a dependently typed core language **DT-Res** (the metalanguage). The application programmer then builds a correct-by-construction application using the DSL.

1.1 Contributions

This paper directly addresses language-based safety and security issues in a Unix-like environment, to illustrate a type-based approach to resource usage verification. *Safety* means that no illegal operations will take place, such as accessing a file handle which has been closed or is not yet opened, or reading from a file handle which is open for writing. *Security* means that only users with appropriate permissions can execute actions.

The main technical contributions of this paper are as follows:

1. We show how to specify and use a resource protocol in a dependently-typed functional language using a DSL as an intermediate step. Since our host language has a sufficiently strong type system, we achieve soundness *without needing to construct specialised correctness proofs*.
2. We show that our approach is *flexible*, in that it is applicable in several resource-aware settings, by applying it separately to file management and concurrent programming problems.
3. We show that our approach is *extensible* and *composable*, in that it is possible to combine the file management and concurrency DSLs into one and guarantee safe concurrent file management.

Furthermore, the underlying type system allows us to specify further security properties such as user permissions on files.

4. We implement a meta-operation, EXECUTE, external to the type theory, which runs *effectful* programs (that is programs that may affect the program environment through I/O or other effects), and show how concrete external data (such as file handles) can be linked with dependently typed programs.

The number of practical examples of dependent types in programming has until recently been very limited. In this paper we show that dependently typed languages can make a valuable contribution to important real world systems by giving novel ways of handling difficult problems. Our implementation is available online[1].

2. Motivating Example: Safe Concurrent Access to a File System

Our primary motivation is to enforce safe program execution in accordance with the constraints of the resource protocols that we wish to deploy. We illustrate this by considering the treatment of race conditions for file accesses in a multi-threaded setting. Consider two concurrent threads, A and B. Each thread would like to append a line to a file `pldi08.txt`, using exactly the same sequence of operations:

1. Read the contents of `pldi08.txt` into a string.
2. Add a new line to the string.
3. Write the resulting string to `pldi08.txt`.

Clearly, the correct outcome is for `pldi08.txt` to be two lines longer after execution of the two threads. If, however, both thread A and thread B read `pldi08.txt` simultaneously, then each thread will add its own new line to the original file contents, ignoring the line that is added by the other thread. `pldi08.txt` will then be only one line longer rather than the two we expect! Naturally, the solution to this race condition is to treat `pldi08.txt` as a shared resource which should be locked throughout the critical sequence.

Such difficulties are, of course, dealt with on an everyday basis by concurrency practitioners, who obtain working solutions by employing a variety of informal coding conventions. However, such conventions are difficult to enforce. Moreover, the use of locking and unlocking operations is potentially highly error-prone, particularly in situations where threads exit under an exception. The (new) approach taken in this paper is to use dependent types to statically enforce the required constraints on the usage of resources such as locks and file handles. In this way, we can prevent the *construction* of programs that could violate these constraints, by ensuring, for example, that race conditions such as the one we have described above can never be encountered.

In our embedded DSL, the above program would be written as:

$$\begin{array}{l}
\mathbf{res} \;=\; (\mathsf{Locked}\,\mathsf{0}\,\mathit{file})::\epsilon \\
\mathbf{addLine} \;:\; \mathsf{CFLang}\;\mathbf{Alice}\;\epsilon\;(\mathsf{Closed}::\epsilon)\;\mathbf{res}\;\mathbf{res} \\
\mathbf{addLine} \;=\; \mathsf{BIND}\;(\mathsf{LOCK}\;\mathsf{f0}\;\Box) \\
\qquad\qquad \lambda_.\;\mathsf{BIND}\;(\mathsf{OPEN}\;\mathit{file}\;\mathsf{Reading}\;\Box\;\Box) \\
\qquad\qquad \lambda h.\;\mathsf{BIND}\;(\mathsf{READ}\;h\;\Box) \\
\qquad\qquad \lambda s.\;\mathsf{BIND}\;(\mathsf{CLOSE}\;h\;\Box) \\
\qquad\qquad \lambda_.\;\mathsf{BIND}\;(\mathsf{OPEN}\;\mathit{file}\;\mathsf{Writing}\;\Box\;\Box) \\
\qquad\qquad \lambda h'.\;\mathsf{BIND}\;(\mathsf{WRITE}\;h'\;\Box\;(s + \text{"}newline\text{"})) \\
\qquad\qquad \lambda_.\;\mathsf{BIND}\;(\mathsf{CLOSE}\;h'\;\Box) \\
\qquad\qquad \lambda_.\;\mathsf{UNLOCK}\;\mathsf{f0}\;\Box
\end{array}$$

where $\Box$ stands for an open proof term, which can be automatically filled in by the type checker when it examines the type context, and which will ensure that the required safety properties will be met. The type of $\mathbf{addLine}$ states that it is written in the embedded DSL CFLang, that there are no file handles on entry, that there is one file handle on exit (and that this is closed), and that all resources are unlocked on entry and exit.

As we will see in Section 7, it would be a compile-time error to omit any of the locking, unlocking or close operations we have specified above, as well as an *automatically checked* runtime error for Alice to open the file if she did not have both read and write permissions to access *file*.

3. Dependently Typed Programming

We will explain our ideas with reference to a simple language, **DT-Res**. **DT-Res** is a basic dependently-typed functional language with the usual properties of subject reduction, Church Rosser, and uniqueness of types, with the additional property that all programs are *total* [28]. In this section we give a brief introduction to the features of **DT-Res** which make it particularly suitable for the implementation of verified Domain-Specific Languages.

3.1 Dependently-Typed Datatypes

In a dependently-typed language, such as **DT-Res**, we can parameterise types over *values*, in addition to the more familiar parameterisation over other types. **DT-Res** is a *full-spectrum* dependently typed language, meaning that *any* value may appear as part of a type, and types may be computed from any value. For example, we may wish define a "lists with length" (or vector) type, where each list has an associated length. In order to do this, we first declare a type of natural numbers, using a GADT-style syntax [22]:

$$\underline{\text{data}}\quad \mathbb{N} : \star \quad \underline{\text{where}} \quad \mathsf{0} : \mathbb{N} \quad | \quad \mathsf{s} : \mathbb{N} \rightarrow \mathbb{N}$$

Here $\mathbb{N}$ is a type[2], with two constructor functions, $\mathsf{0}$ and s. These represent, respectively, a zero, of type $\mathbb{N}$, and the successor function, of type $\mathbb{N} \rightarrow \mathbb{N}$, which takes a value of type $\mathbb{N}$ and returns a new value of type $\mathbb{N}$. Values of type $\mathbb{N}$ may then be used as lengths in the following declaration of vectors:

$$\begin{array}{lll}
\underline{\text{data}} & \mathsf{Vect} : \star \rightarrow \mathbb{N} \rightarrow \star & \underline{\text{where}} \\
& \epsilon \;\; : \mathsf{Vect}\;A\;\mathsf{0} & \\
| & (::) \;\; : A \rightarrow (\mathsf{Vect}\;A\;k) \rightarrow (\mathsf{Vect}\;A\;(\mathsf{s}\;k)) &
\end{array}$$

Note that ϵ only produces vectors of length zero (i.e. of type $\mathsf{Vect}\;A\;\mathsf{0}$), and $x::xs$ only produces vectors whose length is greater than zero (i.e. of type $\mathsf{Vect}\;A\;(\mathsf{s}\;k)$).

The type of Vect, $\star \rightarrow \mathbb{N} \rightarrow \star$, indicates that it is predicated on a type (the element type) and a natural number (the length). When a type includes explicit length information, it follows that a function over that type will express the invariant properties of the length.

In a similar way to our use for vector lengths, we can also use a natural number index to represent *bounds* on sizes. If $\mathsf{Fin}\;n$ is defined to be the set containing precisely n elements:

$$\begin{array}{lll}
\underline{\text{data}} & \mathsf{Fin} : \mathbb{N} \rightarrow \star & \underline{\text{where}} \\
& \mathsf{f0} : \mathsf{Fin}\;(\mathsf{s}\;k) & | \quad \mathsf{fs} : \mathsf{Fin}\;k \rightarrow \mathsf{Fin}\;(\mathsf{s}\;k)
\end{array}$$

we can construct a bounds-safe projection function for vectors:

$$\begin{array}{l}
\underline{\text{let}} \quad \mathbf{lookup} : \mathsf{Fin}\;n \rightarrow \mathsf{Vect}\;A\;n \rightarrow A \\
\quad \mathbf{lookup}\;\;\mathsf{f0}\;\;(x :: ys) \;\mapsto\; x \\
\quad \mathbf{lookup}\;(\mathsf{fs}\;j)\;(x :: ys) \;\mapsto\; \mathbf{lookup}\;j\;ys
\end{array}$$

It is occasionally useful to pair a datatype with its index. In order to do this, we can define a **dependent pair** type, where the type of the second element depends on the value of the first element:

[1] URL removed to preserve anonymity, contact PC chair for details

[2] we write $\mathbb{N} : \star$ to indicate that $\mathbb{N}$ belongs to the type of types, $\star$.

$$\underline{\text{data}}\quad \mathsf{pair} : (A : \star) \to (P : A \to \star) \to \star$$
$$\mathsf{ex} : (a : A) \to P\ a \to \Sigma\ A\ P$$

As a more convenient and readable shorthand, we will write such pair types as Σ-bindings, and values as tuple notation, i.e.:

$$\Sigma n : \mathbb{N}.\ \mathsf{Fin}\ n \quad \text{for} \quad \mathsf{pair}\ \mathbb{N}\ (\lambda n{:}\mathbb{N}.\ \mathsf{Fin}\ n)$$
$$(n, i) \quad \text{for} \quad \mathsf{ex}\ n\ i$$

3.2 Expressing Properties and Predicates on Types

The dependent type system of **DT-Res** allows formal properties to be expressed directly in the types, i.e. as *data* rather than code. For example, consider the following definition of lists over some element type A.

$$\underline{\text{data}}\quad \mathsf{List} : \star \to \star \quad \underline{\text{where}}$$
$$\mathsf{nil} : \mathsf{List}\ A \quad | \quad \mathsf{cons} : A \to \mathsf{List}\ A \to \mathsf{List}\ A$$

we can write a predicate to determine whether a given value is an element of a List:

$$\underline{\text{data}}\quad \mathsf{Elem} : A \to \mathsf{List}\ A \to \star \quad \underline{\text{where}}$$
$$\mathsf{first} : \mathsf{Elem}\ x\ (\mathsf{cons}\ x\ xs)$$
$$|\ \mathsf{later} : \mathsf{Elem}\ x\ ys \to \mathsf{Elem}\ x\ (\mathsf{cons}\ y\ ys)$$

We can view an instance of this predicate as describing the location of an item in a list. Constructing such predicates is most easily achieved by *type-directed* editing, using a tactic based theorem prover [3], similar to COQ [7] or EPIGRAM [20].

3.3 I/O programs

We require input/output operations to be executable from within our type theory. To achieve this, we use Hancock and Setzer's I/O model [9]. This involves implementing a Haskell-style I/O monad [23] by defining *commands*, representing externally executed I/O operations, and *responses* which give the type of the value returned by a given command. Our IO monad is then parametrised over sets of commands and responses:

$$\underline{\text{data}}\quad \mathsf{IO} : (C : \star) \to (R : C \to \star) \to \star \to \star$$
$$\underline{\text{where}}$$
$$\mathsf{Return} : A \to \mathsf{IO}\ C\ R\ A$$
$$|\ \mathsf{Do} : (c : C) \to (R\ c \to \mathsf{IO}\ C\ R\ A) \to \mathsf{IO}\ C\ R\ A$$

Here, Do takes a command, c and an I/O operation that transforms the response to that command into an IO value, and returns the result of applying the I/O operation to the command; and Return simply returns an action packaged as an IO value. To show how this works in practice, we can define simple commands and responses for reading and writing to standard input and output channels. There are two commands:

$$\underline{\text{data}}\quad \mathsf{C_s} : \star \quad \underline{\text{where}}$$
$$\mathsf{ReadStr} : \mathsf{C_s}$$
$$|\ \mathsf{WriteStr} : \mathsf{String} \to \mathsf{C_s}$$

The responses to these commands are a String, in the case that we have read a string, or the unit type if we have written a string.

$$\underline{\text{let}}\quad \mathbf{R_s} : \mathsf{C_s} \to \star$$
$$\mathbf{R_s}\ \mathsf{ReadStr} \mapsto \mathsf{String}$$
$$\mathbf{R_s}\ (\mathsf{WriteStr}\ s) \mapsto \mathsf{Unit}$$

Unit is a single-constructor data type, corresponding to the empty tuple () in e.g. Haskell or ML:

$$\underline{\text{data}}\quad \mathsf{Unit} : \star \quad \underline{\text{where}} \quad \mathsf{unit} : \mathsf{Unit}$$

We can now write the higher-level reading and writing operations:

$$\underline{\text{let}}\quad \mathbf{readStr} : \mathsf{IO}\ \mathbf{C_s}\ \mathbf{R_s}\ \mathsf{String}$$
$$\mathbf{readStr} = \mathsf{Do}\ \mathsf{ReadStr}\ (\lambda s{:}\mathsf{String}.\ \mathsf{Return}\ s)$$
$$\underline{\text{let}}\quad \mathbf{writeStr} : \mathsf{String} \to \mathsf{IO}\ \mathbf{C_s}\ \mathbf{R_s}\ \mathsf{Unit}$$
$$\mathbf{writeStr}\ s = \mathsf{Do}\ (\mathsf{WriteStr}\ s)\ (\lambda x{:}\mathsf{Unit}.\ \mathsf{Return}\ x)$$

Execution of an I/O program consists of evaluating it as normal and passing the result to an EXECUTE meta-operation. This is defined externally to the type theory and simply executes the I/O action at the head of the term, then evaluates and executes the continuation. Execution of the above simple string I/O language is defined by the following pseudo-Haskell code (for convenience, we wil use Haskell-style do notation, with the obvious translation into our Do and Return operations):

```
EXECUTE (Do c r)       ==> RUN c r
EXECUTE (Return v)     ==> return v
RUN ReadStr k          ==> do str <- getLine
                              EXECUTE (k str)
RUN (WriteStr s) k     ==> do putStr s
                              EXECUTE (k unit)
```

4. Unix-style I/O

We aim to implement an embedded DSL supporting *verified* Unix-like I/O operations, for which all the following properties hold:

1. All files are opened before being accessed. Specifically, a program can only read from a file that has been explicitly opened for reading, and can only write to a file that has been explicitly opened for writing.
2. All files are closed before a program exits.
3. A file cannot be closed more than once.
4. File accesses by a user are allowed only if the file permissions allow it. For example, a user cannot write to a file owned by another user unless the file has write permission for other users.

We will define suitable commands and responses for verified file accesses under Unix-like ownership and permissions.

A file can have read, write or execute permission:

$$\underline{\text{data}}\quad \mathsf{Perm} : \star \quad \underline{\text{where}}$$
$$\mathsf{Read} : \mathsf{Perm}\ |\ \mathsf{Write} : \mathsf{Perm}\ |\ \mathsf{Execute} : \mathsf{Perm}$$

It is represented by a record containing its path and the name of its owner (the representations of Filepath and Username are not important, and we can assume that they are strings). It also records details of user permissions (i.e., the operations the owner is allowed to execute), group permissions (i.e., the operations that any user in the group are allowed to execute) and global permissions (i.e. the operations that any user is allowed to execute[3])

$$\underline{\text{data}}\quad \mathsf{File} : \star \quad \underline{\text{where}}$$
$$\mathsf{file} : (name : \mathsf{Filepath}) \to (owner : \mathsf{Username}) \to (group : \mathsf{GroupData}) \to (u : \mathsf{List\ Perm}) \to (g : \mathsf{List\ Perm}) \to (a : \mathsf{List\ Perm}) \to \mathsf{File}$$

A file also records group data, which is simply a reference to the list of users in the relevant group:

$$\underline{\text{data}}\quad \mathsf{GroupData} : \star \quad \underline{\text{where}}$$
$$\mathsf{group} : \mathsf{GroupName} \to \mathsf{List\ Username} \to \mathsf{GroupData}$$

This completes our basic representation of files in a file system. We have not used any dependent types yet, just simply-typed records containing raw file data. We will show how to use this basic representation to implement verified file management in three stages:

[3] While this is not strictly the same as Unix, where global permissions refer to all users *other than* the owner/group, this simplifies our presentation.

1. Confirmation that a user is allowed to access a file for a given purpose (file accessibility).
2. Implementation of I/O operations for file handling. This stage also guarantees that there is no username spoofing; i.e. that once a user is authenticated there is no way to pretend to be another user. The I/O operations rely on the confirmation of accessibility from the first stage.
3. Implementation of the embedded DSL to manage file states, in particular managing when a user is able to open, close and access files.

4.1 File Accessibility

In the first stage, we check whether a user is able to access a file and, if so, we construct a proof of accessibility. First, however, we recap an important technique in dependently-typed programming.

Digression — informative testing

In a simply-typed programming language, testing a value is completely dynamic. Consider the following example from the EPIGRAM tutorial [19]:

```
if null xs then tail xs else xs
```

There is clearly an error here (taking the tail of a list which is known to be empty) but it is not a *type* error, since as far as the type checker is concerned, both branches after the test will correctly return a list. While the result of the test is not known statically, we can easily see from the program that, in the then-branch, `null xs` is always true, that is that `xs` must be the empty list, nil. The problem is that the type for the result of **null** is not sufficiently informative: it is merely a truth value, rather than an explanation of this truth value. If we define a new *view* [20] of lists, we can avoid the error, however:

$$\begin{array}{l} \underline{\text{data}} \quad \mathsf{Null} : \mathsf{List}\ A \to \star \quad \underline{\text{where}} \\ \qquad \mathsf{isNull} : \mathsf{Null}\ \mathsf{nil} \\ \quad \mid \ \mathsf{isCons} : \mathsf{Null}\ (\mathsf{cons}\ x\ xs) \\ \underline{\text{let}} \quad \mathbf{null} : (xs : \mathsf{List}\ A) \to \mathsf{Null}\ xs \end{array}$$

Giving this new type to **null** means that the value is retained in the type rather than dropped following the use of **null**. Consequently, we can determine statically whether the constructor that was tested by **null** was nil or cons simply by looking at the type of the value.

File accessibility predicate

Similarly, when we test whether a file is accessible by a given user, we would like to retain the explanation of *why* it is accessible as part of the result type, and use this when executing any subsequent operations on that file. We can define a predicate which states that a file is accessible for a given purpose by a specific user.

$$\begin{array}{l} \underline{\text{data}} \quad \mathsf{FAcc} : \mathsf{Perm} \to \mathsf{Username} \to \mathsf{File} \to \star \quad \underline{\text{where}} \\ \quad \mathsf{GlobalAcc} : (inPerms : \mathsf{Elem}\ p\ a) \to \\ \qquad\qquad \mathsf{FAcc}\ p\ user\ (\mathsf{file}\ name\ owner\ group\ u\ g\ a) \\ \mid \ \mathsf{UserAcc} : (inPerms : \mathsf{Elem}\ p\ u) \to \\ \qquad\qquad \mathsf{FAcc}\ p\ user\ (\mathsf{file}\ name\ user\ group\ u\ g\ a) \\ \mid \ \mathsf{GroupAcc} : (inPerms : \mathsf{Elem}\ p\ g) \to \\ \qquad\qquad (inGroup : \mathsf{Elem}\ user\ group) \to \\ \qquad\qquad \mathsf{FAcc}\ p\ user\ (\mathsf{file}\ name\ owner \\ \qquad\qquad\qquad (\mathsf{GroupData}\ groupname\ group)\ u\ g\ a) \end{array}$$

Figure 2. File accessibility predicate

Figure 2 shows the FAcc family which represents the possible proofs of file accessibilty. A value $x : \mathsf{FAcc}\ p\ u\ f$ can be viewed as a *proof* or *certificate* that user u is allowed to access file f for the purpose p, for one of the following reasons:

- The permission p is in the list of global permissions for file f, in which case x can be constructed with GlobalAcc.
- The permission p is in the list of user permissions for f *and* the user is the owner of the file, in which case x can be constructed with UserAcc. Note in this case that the variable $user$ is repeated in the type of UserAcc.
- The permission p is in the list of group permissions, and the user is a member of the correct group, in which case x can be constructed with GroupAcc.

Each of the constructors GlobalAcc, UserAcc, or GroupAcc, contains a full explanation of why the file is accessible. Moreover, once we have the value x we always know statically that accessibility has been checked. It follows that not only is it *safe* to execute actions under the accessibility criteria, but we also avoid any possibility of redundant checks.

4.2 File Handles

The second stage of our implementation is to define type-safe I/O operations. We want file operations to be executed only if the file is accessible by the appropriate user, and only also if the file is in an appropriate state (e.g. it can only be closed if it is currently open).

To begin, we will give informative types to file handles. A handle either refers to a file that is open for reading or for writing[4], or it refers to a closed file.

$$\begin{array}{l} \underline{\text{data}} \quad \mathsf{FileState} : \star \quad \underline{\text{where}} \\ \quad \mathsf{Open} : \mathsf{Mode} \to \mathsf{FileState} \ \mid \ \mathsf{Closed} : \mathsf{FileState} \end{array}$$

$$\begin{array}{l} \underline{\text{data}} \quad \mathsf{Mode} : \star \quad \underline{\text{where}} \\ \quad \mathsf{Reading} : \mathsf{Mode} \ \mid \ \mathsf{Writing} : \mathsf{Mode} \end{array}$$

In order to give informative types to file handles, we index them over the possible file states. If a handle is open, we also have a concrete file handle (using a primitive type Handle, defined externally to the type theory):

$$\begin{array}{l} \underline{\text{data}} \quad \mathsf{FileHandle} : \mathsf{FileState} \to \star \quad \underline{\text{where}} \\ \qquad \mathsf{OpenH} : \mathsf{FAcc}\ (\mathbf{reqPerm}\ m)\ u\ fd \to \\ \qquad\qquad \mathsf{Handle} \to \mathsf{FileHandle}\ (\mathsf{Open}\ m) \\ \quad \mid \ \mathsf{ClosedH} : \mathsf{FileHandle}\ \mathsf{Closed} \end{array}$$

The **reqPerm** function computes the permissions that are required for a given file mode:

$$\begin{array}{l} \underline{\text{let}} \quad \mathbf{reqPerm} : \mathsf{Mode} \to \mathsf{Perm} \\ \quad \mathbf{reqPerm}\ \mathsf{Reading} \mapsto \mathsf{Read} \\ \quad \mathbf{reqPerm}\ \mathsf{Writing} \mapsto \mathsf{Write} \end{array}$$

By separating the types of open and closed files and using the state as an index, we can write operations which will only run on file handles in the appropriate state, while still being able to write functions generically over handles where the state is not important. For example, we could define file open, close, read and write operations as follows:

$$\begin{array}{l} \mathbf{open} : \mathsf{String} \to (p : \mathsf{Mode}) \to \mathsf{IO}\ (\mathsf{FileHandle}\ (\mathsf{Open}\ p)) \\ \mathbf{close} : \mathsf{FileHandle}\ (\mathsf{Open}\ p) \to \mathsf{IO}\ (\mathsf{FileHandle}\ \mathsf{Closed}) \\ \mathbf{read} : \mathsf{FileHandle}\ (\mathsf{Open}\ \mathsf{Reading}) \to \mathsf{IO}\ \mathsf{String} \\ \mathbf{write} : \mathsf{String} \to \mathsf{FileHandle}\ (\mathsf{Open}\ \mathsf{Writing}) \to \mathsf{IO}\ \mathsf{Unit} \end{array}$$

Our definitions do not, however, fully satisfy the properties we outlined at the start of this section. There are two problems we need to address: state and security.

[4] for simplicity, we ignore other possibilities here, such as read/write.

Problem 1: State

DT-Res is statically typed, and our types depend on values which may be dynamic. Consider the following code using the operations defined above:

$$\begin{aligned}\mathbf{broken} \;=\; \underline{\text{do}}\; & h \leftarrow \mathbf{open}\, \text{"}\mathtt{testfile}\text{"}\ \mathsf{Reading} \\ & str \leftarrow \mathbf{read}\, h \\ & h' \leftarrow \mathbf{close}\, h \\ & str' \leftarrow \mathbf{read}\, h\end{aligned}$$

This code is type correct, but it contains an error: since h has just been closed, the final read is unsafe. Unfortunately, the type system does not prevent this. In general, we need to be able to deal with the *state*, and therefore the *index on the type* of a value changing.

Problem 2: Security

The reading and writing operations in their simple form do not take into account whether it is valid for a particular user to open a file. We need a more informative type for the **open** operation to ensure that the file is accessible by the user, and more informative types for **read** and **write** to ensure that the correct user is executing the operations. We will address the security problem in Section 5, and then return to the state problem in Section 6.

5. User-indexed I/O operations

In this section we address the security problem detailed above by specialising the IO monad to depend on the user which is executing the I/O operations.

5.1 The UserIO family

We need to ensure that all operations are carried out by a user who has appropriate permissions. Once a user is authenticated, all operations should be carried out by that user, and not by some other, possibly unauthenticated, user. To achieve this, all operations must be *user preserving*. Here we begin to exploit the full power of dependent types. Usernames are run-time values, but by lifting them into types, we can introduce invariant properties of operations which relate to usernames. We define a family of monads, UserIO:

$$\begin{aligned}&\underline{\text{data}}\quad \mathsf{UserIO} \;:\; \mathsf{Username} \rightarrow \star \rightarrow \star \quad \underline{\text{where}} \\ &\quad \mathsf{uio} \;:\; \mathsf{IO}\ \mathsf{C_{fs}}\ \mathbf{R_{fs}}\ A \rightarrow \mathsf{UserIO}\ u\ A\end{aligned}$$

The UserIO type is essentially just a wrapper around the IO monad of Section 3.3, except that the result type for uio also specifies the user, u who is executing the operation. $\mathsf{C_{fs}}$ and $\mathbf{R_{fs}}$ are commands and responses for our file-handling DSL in Section 6.

The most important primitive operation is **bind**. This is used to compose two or more operations of type UserIO into a sequence of operations. **bind** takes two arguments: a UserIO operation on some type A; and a function that takes a value of type A and performs some other UserIO operation on type B. It returns the result of applying the second argument to the result of the first argument. For example,

$$\begin{aligned}\mathbf{readTest} \;=\; & \mathbf{bind}\, (\mathbf{open}\, \text{"}\mathtt{testfile}\text{"}\ \mathsf{Reading}) \\ & (\lambda h.\, \mathbf{read}\, h) \ldots\end{aligned}$$

The type of **bind** is then as follows:

$$\begin{aligned}\underline{\text{let}}\quad \mathbf{bind} \;:\; & \mathsf{UserIO}\ u\ A \rightarrow \\ & (A \rightarrow \mathsf{UserIO}\ u\ B) \rightarrow \mathsf{UserIO}\ u\ B\end{aligned}$$

Since u is passed from one operation to the next, if **bind** is the only interface to the monad, then it is not possible for the username to be changed during the execution of a sequence of operations. We adopt Haskell style <u>do</u> notation which allows the following more readable code, which is identical to the definition of **readTest** above:

$$\begin{aligned}\mathbf{readTest} \;=\; \underline{\text{do}}\; & h \leftarrow \mathbf{open}\, \text{"}\mathtt{testfile}\text{"}\ \mathsf{Reading} \\ & \mathbf{read}\, h \ldots\end{aligned}$$

5.2 Secure File operations in UserIO

We now consider secure, user-preserving IO operations, extending the basic operations of Section 4.2. The types of each of our file handling operations ensure that a user has the proper authorisation. Furthermore, they ensure that once a file is opened for reading, it cannot be written to, and vice versa. For example, opening a file securely requires an accessibility proof:

$$\begin{aligned}\mathbf{openSec} \;:\; & (fd \;:\; \mathsf{File}) \rightarrow (m \;:\; \mathsf{Mode}) \rightarrow \\ & (acc \;:\; \mathsf{FAcc}\ (\mathbf{reqPerm}\ m)\ u\ fd) \rightarrow \\ & \mathsf{UserIO}\ u\ (\mathsf{FileHandle}\ (\mathsf{Open}\ m))\end{aligned}$$

The type states that user u can open a file in mode m if and only if they can provide a proof that they are allowed to do so.

Reading or writing a file is now only allowed on an open file, which can only happen as a result of executing **openSec**. The accessibility proof can therefore be omitted for these operations:

$$\begin{aligned}\mathbf{readSec} \;&:\; \mathsf{FileHandle}\ (\mathsf{Open\ Reading}) \rightarrow \mathsf{UserIO}\ u\ \mathsf{String} \\ \mathbf{writeSec} \;&:\; \mathsf{FileHandle}\ (\mathsf{Open\ Writing}) \rightarrow \mathsf{String} \rightarrow \\ &\quad\; \mathsf{UserIO}\ u\ \mathsf{Unit}\end{aligned}$$

Finally, closing a file makes sense only if a file is already open. The operation does not depend on the file mode:

$$\mathbf{closeSec} \;:\; \mathsf{FileHandle}\ (\mathsf{Open}\ m) \rightarrow \mathsf{UserIO}\ u\ \mathsf{Unit}$$

6. A File Handling DSL

Having used accessibility proofs to solve the security problem, we now turn our attention to state. Since **DT-Res** is purely functional (and therefore stateless), we need to find a mechanism to encapsulate file states in **DT-Res**. Our solution is to embed a file management DSL (the object language) within **DT-Res** (the metalanguage). All file access must then be expressed in terms of this DSL. Our embedded DSL encodes the file operations, manages state, and statically ensures that we do not attempt to violate security or resource properties. Our general method is:

1. Identify the required operations in the DSL, and their types.
2. Write down a corresponding data structure, using dependent types to enforce the types of the required operations and any side conditions.
3. Write an interpreter for the DSL (the object language).

Programs in the DSL have the form $p \;:\; T$, as described by the syntax of Figure 3. The typing rules for the DSL (Figure 4) provide the static guarantees we require that resources and state are managed correctly. This type system is deliberately very simple since we intend to exploit the host language, **DT-Res**, to enforce the side conditions on the rules that guarantee accessibility etc.

We do not write in this language directly, but rather use it in the form of a DSL embedded in **DT-Res**; in the rest of this section we show how to implement it.

$$\begin{aligned}T &::= \mathsf{String} \mid \mathsf{Unit} \mid \mathsf{Handle} \\ x &::= \langle\text{variable}\rangle \mid \langle\text{string}\rangle \\ s &::= \mathsf{OPEN}\ x \mid \mathsf{CLOSE}\ x \mid \mathsf{READ}\ x \mid \mathsf{WRITE}\ x_1\ x_2 \\ p &::= s \mid x \leftarrow p; p \mid p; p\end{aligned}$$

Figure 3. Syntax of the File Handling DSL

$$\frac{\Gamma \vdash x \ : \ \mathsf{String}}{\Gamma \vdash \mathsf{OPEN}\ x \ : \ \mathsf{Handle}}[x \text{ accessible}]$$

$$\frac{\Gamma \vdash x \ : \ \mathsf{Handle}}{\Gamma \vdash \mathsf{CLOSE}\ x \ : \ \mathsf{Unit}}[x \text{ open}]$$

$$\frac{\Gamma \vdash x \ : \ \mathsf{Handle}}{\Gamma \vdash \mathsf{READ}\ x \ : \ \mathsf{Handle}}[x \text{ open for reading}]$$

$$\frac{\Gamma \vdash x \ : \ \mathsf{Handle}}{\Gamma \vdash \mathsf{WRITE}\ x \ : \ \mathsf{Unit}}[x \text{ open for writing}]$$

$$\frac{x \ : \ T \in \Gamma}{\Gamma \vdash x \ : \ T} \qquad \frac{\Gamma \vdash p \ : \ T \qquad \Gamma; x \ : \ T \vdash p' \ : \ T'}{\Gamma \vdash x \leftarrow p; p' \ : \ T}$$

Figure 4. Typing Rules for the File Handling DSL

6.1 List management

We will need a number of helper functions and predicates for managing state and keeping track of invariants in the DSL. Firstly, ElemIs is a predicate on vectors which states the value of an element at a given position. We can think of this as the result of an informative lookup operation, which not only returns a value but also states its relation to the rest of the vector.

$$\begin{array}{lll} \underline{\text{data}} & \mathsf{ElemIs} \ : \ \mathsf{Fin}\ n \to A \to \mathsf{Vect}\ A\ n \to \star & \underline{\text{where}} \\ & \mathsf{atHead} \ : \ \mathsf{ElemIs}\ \mathsf{f0}\ x\ (x{::}xs) & \\ \mid & \mathsf{inTail} \ : \ (y \ : \ A) \to (p \ : \ \mathsf{ElemIs}\ i\ x\ xs) \to & \\ & \qquad \mathsf{Elem}\ (\mathsf{fs}\ i)\ x\ (y{::}\ xs) & \end{array}$$

If an element is in the vector, then it is either at the head (atHead) or else appears in the tail (inTail). Another useful operation is **addEnd**, which adds elements to the end of a vector. This has the invariant that the result is one element longer than the argument.

$$\begin{array}{ll} \underline{\text{let}} & \mathbf{addEnd} \ : \ \mathsf{Vect}\ A\ n \to A \to \mathsf{Vect}\ A\ (\mathsf{s}\ n) \\ \underline{\text{let}} & \mathbf{update} \ : \ \mathsf{Fin}\ n \to A \to \mathsf{Vect}\ A\ n \to \mathsf{Vect}\ A\ n \end{array}$$

6.2 Type and State Representation

A standard technique for implementing type-safe interpreters in dependently-typed languages [5, 21, 6] is to represent an environment as a list, and variables as bounds-safe *de Bruijn* indices [8] into that list. In our implementation, we take a similar approach to managing the file state — we represent the current state as a vector containing all file handles used to date, and treat file handles as references into this list. It is important to note, however, that the user (i.e. the application programmer) does not need to know this detail.

Types in our DSL are either strings, the unit type, or file handles. In the case of file handles we carry the index in the type, since this will be necessary for verifying invariants later. Types in the object language are either strings, the unit type, or file handles. In the case of file handles we carry the index in the type, as this will be necessary for verifying invariants later.

$$\begin{array}{lll} \underline{\text{data}} & \mathsf{FSType} \ : \ \star & \underline{\text{where}} \\ & \mathsf{FString} \ : \ \mathsf{FSType} & \\ \mid & \mathsf{FUnit} \ : \ \mathsf{FSType} & \\ \mid & \mathsf{FHandle} \ : \ \mathsf{Fin}\ n \to \mathsf{FSType} & \end{array}$$

For convenience, we wrap file handles in a dependent pair, so that a user can access a file handle without needing to know the underlying index into the state:

$$\mathbf{fileH} \ = \ \lambda h{:}\mathsf{Fin}\ n.\ \Sigma i \ : \ \mathsf{Fin}\ n.\ (i \ = \ h)$$

We implement a type-level interpretation function for converting object language types into types in the metalanguage:

$$\begin{array}{lll} \underline{\text{let}} \ \ \mathbf{interpTy} \ : \ \mathsf{FSType} \to \star & & \\ \mathbf{interpTy}\ \ \mathsf{FString} & \mapsto & \mathsf{String} \\ \mathbf{interpTy}\ \ \mathsf{FUnit} & \mapsto & \mathsf{Unit} \\ \mathbf{interpTy}\ (\mathsf{FHandle}\ i) & \mapsto & \mathbf{fileH}\ i \end{array}$$

6.3 Representation of the File Handling DSL in DT-Res

Figure 5 shows how our DSL syntax and typing rules can be represented in **DT-Res**. We use one constructor for each of the operations OPEN, CLOSE, READ and WRITE, plus a further constructor for the sequencing operator BIND. Each of these constructors corresponds to one of the typing rules from Figure 4. A file handling program is understood to be run by a particular user, and to transform the incoming file state to an outgoing file state. Therefore, the data type is indexed over these three pieces of information. The constructor for each operation has a type which: i) gives the object language type according to the typing rules; ii) gives the side conditions required for the operation to be executed, and so exposes a proof obligation for the application programmer; and iii) explains how the operation modifies the file state. For example:

$$\begin{array}{l} \mathsf{OPEN} \ : \ (fd \ : \ \mathsf{File}) \to (m \ : \ \mathsf{Mode}) \to \\ \qquad \mathsf{FAcc}\ (\mathbf{reqPerm}\ m)\ u\ fd \to \\ \qquad \mathsf{FSLang}\ u\ hs\ (\mathbf{addEnd}\ hs\ (\mathsf{Open}\ m)) \\ \qquad\qquad (\mathsf{FHandle}\ (\mathbf{top}\ in)) \end{array}$$

Given a filename, a mode and an accessibility proof (which is effectively a security certificate), the OPEN operation is permitted to run. As it does so, it transforms the incoming list of file handles, hs, to a new list, hs', with an open file handle at the end. Adding the handle at the end of the list ensures that an index will always refer to the same file. This avoids needing to correctly weaken indices whenever a new handle is created. **top** gives the new file index, which will be the highest value in Fin in.

Operations which access file handles work by accessing the file state at the time the operation runs. Since the object language representation is indexed over the incoming and outgoing file state, we know precisely what this state is, and can therefore use the ElemIs predicate defined previously to index into the state. For example, if we want to READ from a file, we must be able to guarantee that the handle we are trying to read from is open *at this time*:

$$\begin{array}{l} \mathsf{READ} \ : \ (i \ : \ \mathsf{Fin}\ in) \to \mathsf{ElemIs}\ i\ hs\ (\mathsf{Open\ Reading}) \to \\ \qquad \mathsf{FSLang}\ u\ hs\ hs\ \mathsf{FString} \end{array}$$

The BIND operation sequences actions, and is very similar to a bind operation on a monad (see section 5.1), additionally threading *state*. With this operation, we can safely pass intermediate values and intermediate states through a code sequence and at each stage know exactly what the file state is.

$$\begin{array}{l} \mathsf{BIND} \ : \ (code \ : \ \mathsf{FSLang}\ u\ hs_0\ hs_1\ S) \to \\ \qquad ((\mathbf{interpTy}\ S) \to \mathsf{FSLang}\ u\ hs_1\ hs_2\ T) \to \\ \qquad \mathsf{FSLang}\ u\ hs_0\ hs_2\ T \end{array}$$

There is a major benefit to implementing sequencing this way, using BIND and a continuation, rather than an explicit sequencing operator. Passing variables in the continuation allows us to exploit the **DT-Res** metalanguage to represent variables other than the file states we are managing, so simplifying the implementation.

6.4 Interpreting Operations in the File Handling DSL

To run such programs, we write an intepreter **runFS** which carries an environment of file states, and which returns both the resulting file states and the result of interpretation. The environment contains the actual values of the file handles corresponding to the file states. It is indexed over the vector of file states. This ensures that the environment is type safe:

$$
\begin{array}{lll}
\underline{\text{data}} & \mathsf{FSLang} : \mathsf{Username} \to \mathsf{Vect}\ \mathsf{FileState}\ in \to \mathsf{Vect}\ \mathsf{FileState}\ out \to \mathsf{FSType} \to \star & \underline{\text{where}} \\
& \mathsf{OPEN} : (fd : \mathsf{File}) \to (m : \mathsf{Mode}) \to & \\
& \quad \mathsf{FAcc}\ (\mathbf{reqPerm}\ m)\ u\ fd \to \mathsf{FSLang}\ u\ hs\ (\mathbf{addEnd}\ hs\ (\mathsf{Open}\ m))\ (\mathsf{FHandle}\ (\mathbf{top}\ in)) & \\
\mid & \mathsf{CLOSE} : (i : \mathsf{Fin}\ in) \to \mathsf{ElemIs}\ i\ hs\ (\mathsf{Open}\ m) \to \mathsf{FSLang}\ u\ hs\ (\mathbf{update}\ i\ \mathsf{Closed}\ hs)\ \mathsf{FUnit} & \\
\mid & \mathsf{READ} : (i : \mathsf{Fin}\ in) \to \mathsf{ElemIs}\ i\ hs\ (\mathsf{Open\ Reading}) \to \mathsf{FSLang}\ u\ hs\ hs\ \mathsf{FString} & \\
\mid & \mathsf{WRITE} : (i : \mathsf{Fin}\ in) \to \mathsf{ElemIs}\ i\ hs\ (\mathsf{Open\ Write}) \to \mathsf{String} \to \mathsf{FSLang}\ u\ hs\ hs\ \mathsf{FUnit} & \\
\mid & \mathsf{BIND} : (code : \mathsf{FSLang}\ u\ hs_0\ hs_1\ S) \to & \\
& \quad ((\mathbf{interpTy}\ S) \to \mathsf{FSLang}\ u\ hs_1\ hs_2\ T) \to \mathsf{FSLang}\ u\ hs_0\ hs_2\ T &
\end{array}
$$

Figure 5. Representation of the File Handling DSL in **DT-Res**

$$
\begin{array}{ll}
\underline{\text{data}} & \mathsf{Env} : \mathsf{Vect}\ \mathsf{FileState}\ n \to \star \quad \underline{\text{where}} \\
& \mathsf{empty} : \mathsf{Env}\ \epsilon \\
\mid & \mathsf{extend} : \mathsf{FileHandle}\ f \to \mathsf{Env}\ hs \to \mathsf{Env}\ (f :: hs)
\end{array}
$$

Projecting a value from an environment is only valid if the file handle we expect to project is in the appropriate state in the environment. We therefore pass an instance of the ElemIs family to **lookupEnv**. This then certifies that the environment contains a file in the correct state. Corresponding to **addEnd** and **update** on vectors, we write **extendEnv** and **updateEnv** on environments. These functions are implemented by pattern matching on the Env:

$$
\begin{array}{lll}
\underline{\text{let}} & \mathbf{lookupEnv} : & \mathsf{Env}\ hs \to \mathsf{ElemIs}\ i\ s\ hs \to \mathsf{FileHandle}\ s \\
\underline{\text{let}} & \mathbf{extendEnv} : & \mathsf{Env}\ hs \to \mathsf{FileHandle}\ s \to \\
& & \mathsf{Env}\ (\mathbf{addEnd}\ hs\ s) \\
\underline{\text{let}} & \mathbf{updateEnv} : & \mathsf{Fin}\ i \to \mathsf{FileHandle}\ s \to \mathsf{Env}\ hs \to \mathsf{Env}\ hs
\end{array}
$$

The interpreter itself, **runFS** is given in Figure 6. Its type corresponds to the type of the object language representation — given a program which takes file states hs_{in} to hs_{out}, **runFS** will take an environment of file handles indexed over hs_{in} and give a new environment of file handles indexed over hs_{out}.

$$
\begin{array}{l}
\mathbf{runFS} : \mathsf{Env}\ hs_{in} \to \mathsf{FSLang}\ u\ hs_{in}\ hs_{out}\ T \to \\
\qquad \mathsf{UserIO}\ u\ (\mathsf{Env}\ hs_{out}, \mathbf{interpTy}\ T) \\
\quad \mathbf{runFS}\ hs\ (\mathsf{OPEN}\ f\ m\ acc) \\
\qquad \mapsto \underline{\text{do}}\ h \leftarrow \mathbf{openSec}\ f\ m\ acc \\
\qquad\qquad \underline{\text{return}}\ (\mathbf{extendEnv}\ hs\ h, \mathbf{bound}\ _) \\
\quad \mathbf{runFS}\ hs\ (\mathsf{CLOSE}\ i\ p) \\
\qquad \mapsto \underline{\text{do}}\ x \leftarrow \mathbf{closeSec}\ (\mathbf{lookupEnv}\ hs\ p) \\
\qquad\qquad \underline{\text{return}}\ (\mathbf{updateEnv}\ i\ \mathsf{ClosedH}, x) \\
\quad \mathbf{runFS}\ hs\ (\mathsf{READ}\ i\ p) \\
\qquad \mapsto \underline{\text{do}}\ str \leftarrow \mathbf{readSec}\ (\mathbf{lookupEnv}\ hs\ p) \\
\qquad\qquad \underline{\text{return}}\ (hs, str) \\
\quad \mathbf{runFS}\ hs\ (\mathsf{WRITE}\ i\ p\ str) \\
\qquad \mapsto \underline{\text{do}}\ str \leftarrow \mathbf{writeSec}\ (\mathbf{lookupEnv}\ hs\ p)\ str \\
\qquad\qquad \underline{\text{return}}\ (hs, \mathsf{unit}) \\
\quad \mathbf{runFS}\ hs_0\ (\mathsf{BIND}\ code\ k) \\
\qquad \mapsto \underline{\text{do}}\ (hs_1, c_1) \leftarrow \mathbf{runFS}\ code\ hs_0 \\
\qquad\qquad \mathbf{runFS}\ (k\ c_1)\ hs_1
\end{array}
$$

Figure 6. The DSL Interpreter

6.5 Constructing proofs

In order to use the file handling DSL, the application programmer must construct proofs of file accessibility and proofs that a handle is available. Fortunately, these proofs are largely automatable since both accessibility and list membership are decidable.

$$
\begin{array}{l}
\mathbf{accessible} : (p : \mathsf{Perm}) \to (u : \mathsf{Username}) \to \\
\qquad (f : \mathsf{File}) \to \mathsf{Maybe}\ (\mathsf{FAcc}\ p\ u\ f) \\
\mathbf{elem} : (i : \mathsf{Fin}\ n) \to (a : A) \to \\
\qquad (xs : \mathsf{Vect}\ A\ n) \to \mathsf{Maybe}\ (\mathsf{ElemIs}\ i\ a\ xs)
\end{array}
$$

Since we are using a type-directed development environment, the DSL author can easily provide a tactic for filling in these proofs. A tactic `decide` runs one of the above functions and fills in the resulting proof, or reports if no proof can be found — if there is no proof, the operation the programmer is trying to execute is invalid.

7. Concurrency

7.1 A Concurrent Lock-Handling DSL

As multi-core architectures become more common, it becomes increasingly important to manage concurrent access to system resources in an efficient and effective manner. In order to do this, it is necessary to manage concurrent locks on shared resources. In this section, we apply the same method we have used for file handles to *nested* locks. We introduce resource states, analogous to file states, which count the number of times a given resource has been locked. We will make this polymorphic over the locked resource:

$$
\begin{array}{ll}
\underline{\text{data}} & \mathsf{ResourceState} : \star \to \star \quad \underline{\text{where}} \\
& \mathsf{Locked} : \mathbb{N} \to A \to \mathsf{ResourceState}\ A
\end{array}
$$

Corresponding to file handles, we have resource handles, which relate locks to some concrete data that is being locked and an externally defined implementation of locks, Lock:

$$
\begin{array}{ll}
\underline{\text{data}} & \mathsf{ResourceHandle} : (A : \star) \to (\mathsf{ResourceState}\ A) \to \star \\
& \underline{\text{where}} \\
& \mathsf{Resource} : A \to \mathsf{Lock} \to \mathsf{ResourceHandle}\ A\ (\mathsf{Locked}\ n)
\end{array}
$$

Our lock-handling DSL supports resource handles, unit types, plus a I/O actions over arbitrary types. These types are defined in the **DT-Res** meta language as follows:

$$
\begin{array}{ll}
\underline{\text{data}} & \mathsf{CType} : \star \quad \underline{\text{where}} \\
& \mathsf{RHandle} : \mathsf{Fin}\ n \to \mathsf{CType} \\
\mid & \mathsf{RUnit} : \mathsf{CType} \\
\mid & \mathsf{RType} : \star \to \mathsf{CType}
\end{array}
$$

When we deal with specific resources, we should be careful that any resource which *must* be locked is encapsulated in a safe API. The DSL that can be used to achieve this, CLang, is shown in Figure 7. The general structure is similar to the file-handling DSL, but with resource handles in place of file handles and omitting username information.

In order to implement the interpreter, we follow a similar approach to that previously taken for the file-handling DSL. The interpreter already knows about the resources it can lock and unlock. It is free to create concurrent threads as required using operating system services. The interpreter follows the structure of **runFS** for file systems, calling underlying operating-system functions for

$$
\begin{array}{lll}
\underline{\text{data}} & \mathsf{CLang} : \mathsf{Vect}\,(\mathsf{ResourceState}\ A)\ in \to \mathsf{Vect}\,(\mathsf{ResourceState}\ A)\ out \to \mathsf{CType} \to \star \quad \underline{\text{where}} \\
& \mathsf{LOCK} : (i : \mathsf{Fin}\ in) \to \mathsf{Elemls}\ i\ hs\ (\mathsf{Locked}\ n) \to \mathsf{CLang}\ hs\ (\mathbf{update}\ i\ (\mathsf{Locked}\ a\ (\mathsf{s}\ n)))\ \mathsf{RUnit} \\
| & \mathsf{UNLOCK} : (i : \mathsf{Fin}\ in) \to \mathsf{Elemls}\ i\ hs\ (\mathsf{Locked}\ (\mathsf{s}\ n)) \to \mathsf{CLang}\ hs\ (\mathbf{update}\ i\ (\mathsf{Locked}\ a\ n))\ \mathsf{RUnit} \\
| & \mathsf{ACTION} : (\mathsf{IO}\ T) \to \mathsf{CLang}\ hs\ hs\ (\mathsf{RType}\ T) \\
| & \mathsf{FORK} : (thread : \mathsf{CLang}\ hs\ hs\ \mathsf{RUnit}) \to \mathsf{CLang}\ hs\ hs\ \mathsf{RUnit} \\
| & \mathsf{BIND} : (code : \mathsf{CLang}\ u\ hs_0\ hs_1\ S) \to ((\mathbf{interpTy}\ S) \to \mathsf{CLang}\ u\ hs_1\ hs_2\ T) \to \mathsf{CLang}\ u\ hs_0\ hs_2\ T
\end{array}
$$

Figure 7. The Concurrent Lock-Handling DSL

locking and unlocking handles and forking threads. For brevity, we show only the type of the interpreter here. The complete implementation is available online[5].

$$
\begin{array}{ll}
\mathbf{runC} : & \mathsf{Env}\ hs_{in} \to \mathsf{FSLang}\ u\ hs_{in}\ hs_{out}\ T \to \\
& \mathsf{IO}\,(\mathsf{Env}\ hs_{out}, \mathbf{interpTy}\ T)
\end{array}
$$

Once again, by making this interpreter the *sole* interface to the locking mechanism, we can ensure that all locks are managed correctly, that is that all locks that are acquired will be released by the time the program exits, and that no attempt is made to release a lock which is not taken. The type of the FORK command ensures that a thread releases all locks when it completes. This interpreter is the single point where concurrent lock handling programs can fail — if the interpreter manages resources correctly, any programs that it can evaluate are also correct with respect to resource management.

7.2 Concurrent file handling

We have now implemented two separate DSLs using the same method. The first DSL handles files securely and cleans up open files on exit. The second DSL correctly manages concurrent resource locks. However, files are important examples of resources that may need to be locked under concurrency. We would therefore like to combine our two DSLs into a single concurrent file-handling DSL. Fortunately, since both kinds of resource (locks and files) are largely independent, it is fairly simple to combine these two DSLs.

We choose to construct the combined DSL in such a way that it is not possible to open a file without first acquiring a lock on it. We link a ResourceHandle with the name of a file. Our DSL combines the commands for each DSL into a single set. We also combine the indices: username, file handles *and* resource handles.

$$
\begin{array}{lll}
\underline{\text{data}} & \mathsf{CFLang} : & \mathsf{Username} \to \mathsf{Vect}\ \mathsf{FileState}\ in \to \\
& & \mathsf{Vect}\ \mathsf{FileState}\ out \to \\
& & \mathsf{Vect}\,(\mathsf{ResourceState}\ A)\ in \to \\
& & \mathsf{Vect}\,(\mathsf{ResourceState}\ A)\ out \to \\
& & \mathsf{FSType} \to \star
\end{array}
$$

The OPEN operation now requires a proof that the resource has been locked at least once (i.e. s n times), in addition to a proof that the file is accessible.

$$
\begin{array}{ll}
\mathsf{OPEN} : & (fd : \mathsf{File}) \to (m : \mathsf{Mode}) \to \\
& \mathsf{FAcc}\,(\mathbf{reqPerm}\ m)\ u\ fd \to \\
& \mathsf{Elemls}\ i\ rs\ (\mathsf{Locked}\ (\mathsf{s}\ n)\ fd) \to \\
& \mathsf{CFLang}\ u\ hs\ (\mathbf{addEnd}\ hs\ (\mathsf{Open}\ m)) \\
& \qquad rs\ rs\ (\mathsf{FHandle}\ (\mathbf{top}\ in))
\end{array}
$$

While the type of this may look complicated, an application programmer would merely write

$$\mathsf{OPEN}\ fd\ \mathsf{Writing}\ \Box_1\ \Box_2$$

with $\Box_1$ and $\Box_2$ being completed by proof search tactics as outlined in Section 6.5. The combined DSL is shown in Figure 8. CFType is the obvious combination of FSType and CFType for the file-handling and concurrency DSLs. Note that a thread introduced by FORK may introduce new file handles, but that these are inaccessible from the parent thread.

We can now revisit the motivating example we outlined in Section 2. The implementation of each thread is shown below.

$$
\begin{array}{l}
\mathbf{RW} = \mathsf{cons\ Read\ (cons\ Write\ nil)} \\
\mathbf{R} = \mathsf{cons\ Read\ nil} \\
\mathbf{noPerms} = \mathsf{nil} \\
\mathbf{file} = \mathsf{file}\ \texttt{"pldi08.txt"}\ \mathbf{Alice\ users\ RW\ R\ noPerms} \\
\mathbf{res} = (\mathsf{Locked}\ 0\ \mathbf{file})::\epsilon
\end{array}
$$

$$
\begin{array}{ll}
\mathbf{addLine} : & \mathsf{CFLang}\ \mathbf{Alice}\ \epsilon\ (\mathsf{Closed}::\epsilon)\ \mathbf{res\ res} \\
\mathbf{addLine} = & \mathsf{BIND}\ (\mathsf{LOCK}\ \mathsf{f0}\ \Box) \\
& \lambda_.\ \mathsf{BIND}\ (\mathsf{OPEN}\ \mathbf{file}\ \mathsf{Reading}\ \Box\ \Box) \\
& \lambda h.\ \mathsf{BIND}\ (\mathsf{READ}\ h\ \Box) \\
& \lambda s.\ \mathsf{BIND}\ (\mathsf{CLOSE}\ h\ \Box) \\
& \lambda_.\ \mathsf{BIND}\ (\mathsf{OPEN}\ \mathbf{file}\ \mathsf{Writing}\ \Box\ \Box) \\
& \lambda h'.\ \mathsf{BIND}\ (\mathsf{WRITE}\ h'\ \Box\ (s + \text{"}newline\text{"})) \\
& \lambda_.\ \mathsf{BIND}\ (\mathsf{CLOSE}\ h'\ \Box) \\
& \lambda_.\ \mathsf{UNLOCK}\ \mathsf{f0}\ \Box
\end{array}
$$

$$
\begin{array}{ll}
\mathbf{addLines} = & \mathsf{BIND}\ (\mathsf{FORK}\ \mathbf{addLine}) \\
& \lambda_.\ \mathbf{addLine}
\end{array}
$$

This program sets up a list of shared resources, in this case just one file. Since Alice has both read and write permissions for the file, it follows that the file is *accessible*. The type of **addLine** states that it is a program which begins with no file handles and ends with one file handle, which has been closed. It also states that the state of all the locks on entry and exit is the same. We achieve this by unlocking all of the locks we have acquired. We *must* acquire a lock on *file* before opening it, otherwise the program will not compile. While there are several places in this program which require proof details, as shown by $\Box$, if a tactic-based type-directed editor is used, it is not a problem to complete these details, since the proof objects arise directly from the context. We note that by removing the UNLOCK and LOCK commands, this program would also be valid in the file-handling DSL.

8. Related Work

Previous approaches to resource usage verification have typically been based on developing special purpose type systems, *post-hoc* program analysis, or a combination of these techniques. For example, Marriott et al [18] use a deterministic finite state automaton (DFA) to describe the allowed states of resources. Their approach relies on a program analysis that models the *approximate* behaviour of the program, and that then checks that this behaviour conforms to the DFA. In contrast, in our approach, we effectively place the permissible states in the types of the DSL. This allows us to relate the *real* program, rather than an approximate model, to the permitted behaviour and so to guarantee correctness *by construction* and without the limitations of a DFA. In particular, unlike a DFA, we

[5] URL removed to preserve anonymity, contact PC chair for details

```
data   CFLang : Username → Vect FileState in → Vect FileState out →
                Vect (ResourceState File) in → Vect (ResourceState File) out → CFType → ⋆   where
    OPEN : (fd : File) → (m : Mode) → FAcc (reqPerm m) u fd → ElemIs i rs (Locked (s n) fd) →
                CFLang u hs (addEnd hs (Open m)) rs rs (FHandle (top in))
  | CLOSE : (i : Fin in) → ElemIs i hs (Open m) → CFLang u hs (update i Closed hs) rs rs RUnit
  | READ : (i : Fin in) → ElemIs i hs (Open Reading) → FSLang u hs hs rs rs FString
  | WRITE : (i : Fin in) → ElemIs i hs (Open Write) → String → CFLang u hs hs rs rs RUnit
  | LOCK : (i : Fin in) → ElemIs i rs (Locked n) → CFLang hs hs rs (update i (Locked a (s n))) RUnit
  | UNLOCK : (i : Fin in) → ElemIs i rs (Locked (s n)) → CFLang hs hs rs (update i (Locked a n)) RUnit
  | ACTION : (IO T) → CFLang hs hs rs rs (RType T)
  | FORK : (thread : CFLang hs0 hs1 rs rs RUnit) → CFLang hs0 hs0 rs rs RUnit
  | BIND : (code : CFLang u hs0 hs1 rs0 rs1 S) → ((interpTy S) → CFLang u hs1 hs2 rs1 rs2 T) →
            CFLang u hs0 hs2rs0 rs2 T
```

Figure 8. The Concurrent Locking and File-Handling DSL

are not limited to a predetermined number of states — the states we need (such as username or file permissions) can be decided at run-time. Closer to our work in this sense is [27], which uses scoping rules to prevent some kinds of incorrect file accesses. However, this method is not applicable more generally to security properties such as file accessibility and username invariants as used above.

Igarashi and Kobayashi have also developed a type system [12] for guaranteeing resource properties. This is similar to our approach in that they attempt to provide a uniform approach to solving resource usage problems rather than solving a specific resource problem such as space usage or file handling, but differs in that they develop a new resource type system rather than exploiting an existing general purpose type system as we have done here. They consequently must also develop both new soundness proofs and a new type checking implementation.

An obvious approach to dealing with resources is to use linear types, e.g. [11, 10]. We have avoided this for two reasons: firstly, we believe that linear types are too restrictive for general-purpose programming. Secondly, we believe that dependent types are sufficiently strong to deal with the explicit creating and freeing of resources. We therefore prefer to exploit the properties of a well-understood type system rather than implement a new one, with its consequent requirement for new (and tedious) soundness proofs.

In addition to these general approaches, various special-purpose type systems have also been developed. For example [30] can be used to enforce security properties and [15, 26] can be used to prevent deadlock and race condiitions. While these approaches seem promising as specialised applications, we prefer to build on a strong general-purpose type system and construct our programs in such a way as to allow *composable* domain-specific languages.

An alternative but related approach involves exploiting program monitoring [17] to dynamically check that a program adheres to security constraints and to take remedial action before a dangerous action is executed. Our approach differs from monitoring-based approaches in that we are able to statically guarantee that remedial action will never be necessary. It follows that the overheads of monitoring can be completely avoided in our case.

Finally, while we have not explored this in detail, some parts of our approach may be adaptable to weaker type systems, such as Generalised Algebraic Data Types (GADTs) [22] in Haskell or Omega [25], or perhaps even by enforcing resource constraints using Haskell type classes, as suggested by Kiselyov and Shan [14]. The real benefit we obtain from using full dependent types is the ability to lift values and functions directly, without modification, into types which allows us for example to construct the UserIO monad and FAcc predicate immediately.

9. Conclusions

We have shown how to develop embedded DSLs in a dependently typed language that are capable of statically guaranteeing correct resource usage *by construction*. We have developed two independent DSLs: one for file management in a Unix-like file system and one for lock handling in concurrent programs. In addition, we have shown that DSLs implemented in this way can be *composable*, in this case by combining the constructors and indices from each DSL into a new combined DSL that implements both locking and file handling. Although we have here composed the DSLs by hand, we anticipate that *generic programming* or *program generation* techniques might allow this to be automated to some extent in future.

Specifying some aspects of a program behaviour directly in its type has allowed us to directly derive some important safety properties of programs written in the concurrent file-handling DSL:

- No two threads will access a file simultaneously, since a file cannot be opened until a thread can prove it has obtained a lock.
- No resource can be accessed in an inconsistent state. For example, write operations are not permitted on a file that is currently open only for reading.
- All resources are cleaned up on exit: all locks are released and all open files are closed.

The primary advantage of our approach over previous work is that these properties arise entirely from the type soundness of **DT-Res**. Although we do, in effect, implement simple type systems, we can exploit the existing implementation of the **DT-Res** type checker. We also use the **DT-Res** type system to guarantee that our interpreters are type preserving, so freeing us of much of the burden associated with constructing a new type system. The meta-theoretical properties of **DT-Res** — type preservation, Church-Rosserness, strong normalisation — are by definition also properties of the implemented DSLs. In effect, we have moved from informal coding conventions to manage resource locks to formal checkable conventions. Because we have used dependent types, the only programs we can write in the DSL are necessarily *correct by construction*. In order to verify programs in the DSL, there are no theorems to prove, because the program *is* the theorem [29].

9.1 Further Work

There are two obvious ways in which this work could be developed further. Firstly, we could apply our ideas to other areas with strong safety or security requirements, developing new DSLs as appropriate; and secondly, we could directly extend the DSLs we have given here to cover additional safety or security capabilities.

Moreover, although we have described how to combine two DSLs into a larger DSL for dealing with multiple resource constraints, this new DSL was constructed by hand, rather than automatically. Since the DSLs share several characteristics (in particular, a BIND operation) it would be better to construct a generic language template into which specific details of the DSL could be inserted, in the spirit of Landin's ISWIM [16].

There are a number of areas that would repay further study. Firstly, we have dealt here with one aspect of concurrent programming, namely ensuring that (nested) locks are created before accessing a shared resource and that they are released when the program finishes. However, we have only dealt with one type of lock — we may wish to deal with locks which cannot be nested (which we can do simply by limited the lock count to 1), with semaphores or with shared variables. We would expect to be able to deal with the latter by extending the type of the FORK operation but would then also need to deal with synchronisation and communication between threads.

Secondly, modern operating-system kernels require similar locking operations. On modern multi-core processors it is especially important that resources are managed efficiently to reduce bottlenecks in accessing I/O devices. Our method provides a promising research direction for exploiting modern processors while providing safe abstractions for operating system and device driver programmers.

Thirdly, we plan to extend our implementation to handle more sophisticated security methods, such as Access Control Lists for file systems and networking. The combination of locking resources and access control policies in particular suggest that our method may also be directly applicable to database management.

Fourthly, we believe our approach to be applicable in a number of other contexts. One area we have begun to investigate is network protocol correctness. In this context, one especially interesting empirical measure of the value of the approach would be whether correct-by-construction programs could determine errors in existing protocols.

Finally, one important resource usage problem we have previously looked at using other methods [2], but have not addressed in this paper, is the memory usage of functional programs. While we have previously analysed *heap* usage bounds with dependent types, it has proved more problematic to deal with *stack* bounds, because, unlike data structures or general heap, stacks do not increase in size monotonically throughout a program's execution. The method we have described here seems promising for addressing this problem, however, because of the way the BIND operation provides a mechanism to track the changes in the resource state at each stage of the program's execution.

References

[1] Omitted to preserve anonymity, contact PC chair for details.

[2] Omitted to preserve anonymity, contact PC chair for details.

[3] Omitted to preserve anonymity, contact PC chair for details.

[4] P. Amey. Correctness by Construction: Better can also be Cheaper. *CrossTalk: the Journal of Defense Software Engineering*, pages 24–28, Mar. 2002.

[5] L. Augustsson and M. Carlsson. An exercise in dependent types: A well-typed interpreter, 1999.

[6] E. Brady and K. Hammond. A verified staged interpreter is a verified compiler. In *Proc. Conf. Generative Programming and Component Engineering (GPCE '06)*, 2006.

[7] Coq Development Team. The Coq proof assistant — reference manual. `http://coq.inria.fr/`, 2001.

[8] N. de Bruijn. Lambda calculus notation with nameless dummies. *Indagationes Mathematicae*, 34:381–392, 1972.

[9] P. Hancock and A. Setzer. Interactive programs in dependent type theory. In P. Clote and H. Schwichtenberg, editors, *Proc. of 14th Ann. Conf. of EACSL, CSL'00, Fischbau, Germany, 21–26 Aug 2000*, volume 1862, pages 317–331. Springer-Verlag, Berlin, 2000.

[10] C. Hawblitzel. Linear types for aliased resources. Technical Report MSR-TR-2005-141, Microsoft Research, 2005.

[11] M. Hofmann and S. Jost. Static prediction of heap space usage for first-order function al programs. In *Proc. POPL 2003 — 2003 ACM Symp. on Principles of Programming La nguages*, pages 185–197. ACM, 2003.

[12] A. Igarashi and N. Kobayashi. Resource usage analysis. In *Symposium on Principles of Programming Languages*, pages 331–342, 2002.

[13] N. Jin and J. He. Towards a truly concurrent model for processes sharing resources. In *Proc. 3rd IEEE International Conf. on Soft. Eng. and Formal Methods*, pages 231–239, Washington, DC, USA, 2005. IEEE Computer Society.

[14] O. Kiselyov and C.-C. Shan. Lightweight static resources: Sexy types for embedded and systems programming, 2007. Draft.

[15] N. Kobayashi. A type system for lock-free processes. *Inf. Comput.*, 177(2):122–159, 2002.

[16] P. Landin. The next 700 programming languages. *Communications of the ACM*, 9(3), March 1966.

[17] J. A. Ligatti. *Policy Enforcement via Program Monitoring*. PhD thesis, May 2006.

[18] K. Marriott, P. Stuckey, and M. Sulzmann. Resource usage verification. In *In Proc. of First Asian Symposium, APLAS 2003*, pages 212–229. Springer-Verlag, 2003.

[19] C. McBride. Epigram: Practical programming with dependent types. Lecture Notes, International Summer School on Advanced Functional Programming, 2004.

[20] C. McBride and J. McKinna. The view from the left. *Journal of Functional Programming*, 14(1):69–111, 2004.

[21] E. Pašalíc, W. Taha, and T. Sheard. Tagless staged interpreters for typed languages. In *Proc. 2002 International Conf. on Functional Programming (ICFP 2002)*. ACM, 2002.

[22] S. Peyton Jones, D. Vytiniotis, S. Weirich, and G. Washburn. Simple unification-based type inference for GADTs. In *Proc. 2006 International Conf. on Functional Programming (ICFP 2006)*, 2006.

[23] S. L. Peyton Jones and P. Wadler. Imperative functional programming. In *Proc. 20th ACM Symposium on Principles of programming languages*, pages 71–84, New York, NY, USA, 1993. ACM.

[24] C. Popeea and W.-N. Chin. A type system for resource protocol verification and its correctness proof. In *PEPM '04: Proceedings of the 2004 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, pages 135–146, New York, NY, USA, 2004. ACM Press.

[25] T. Sheard. Languages of the future. In *ACM Conference on Object Orientated Programming Systems, Languages and Applicatioons (OOPSLA'04)*, 2004.

[26] K. Suenaga and N. Kobayashi. Type-based analysis of deadlock fora concurrent calculus with interrupts. 2007.

[27] G. Tan, X. Ou, and D. Walker. Resource usage analysis via scoped methods. In *Foundations of Object-Oriented Languages*, 2003.

[28] D. A. Turner. Elementary strong functional programming. In *First International Symposium on Functional Programming Languages in Education*, volume 1022 of *LNCS*, pages 1–13. Springer, 1995.

[29] P. Wadler. Theorems for free! In *Proc. 4th Int. Conf. on Funct. Prog. Languages and Computer Arch., FPCA'89, London, UK, 11–13 Sept 1989*, pages 347–359. ACM Press, New York, 1989.

[30] D. Walker. A type system for expressive security policies. In *Twenty-Seventh ACM SIGPLAN Symposium on Principles of Programming Languages*, pages 254–267, 2000.