

Correct-by-Construction Concurrency

Edwin Brady Kevin Hammond

School of Computer Science, University of St Andrews, St Andrews, Scotland.

Email: eb,kh@cs.st-andrews.ac.uk

Abstract

In the modern, multi-threaded, multi-core programming environment, correctly managing system resources such as locks and shared variables 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 often only loosely integrated 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 Embedded Language (DSEL) whose typing rules enforce the properties we require. We illustrate our approach by implementing a DSEL for concurrent programming and demonstrate its applicability with reference to concurrent transactions on a bank account.

1. Introduction

As multi-core architectures become more common, so approaches to concurrency based on serialising access to system resources are becoming a major potential performance bottleneck. Truly concurrent access to system resources [16], 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” [2], 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.

Popeea and Chin observe [26] that there are two key facets to verifying resource access protocols: *protocol verification*, to ensure

that the protocol conforms to the required formal properties; 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 [6, 5]. 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 on shared state, to construct a simple dependently-typed Domain-Specific Embedded Language (DSEL). The typing rules of this DSEL then directly enforce the required safety properties, such as safe acquisition and release of resource locks.

1.1 Contributions

This paper directly addresses language-based safety and security issues in a multi-threaded environment with shared state, to illustrate a type-based approach to resource usage verification, addressing both theoretical and practical considerations. The main technical contributions of this paper are as follows:

1. We give a general method for implementing a state-handling domain-specific embedded language in a dependently typed meta-language. (Section 5)
2. We show how to specify and use a resource access protocol in a dependently typed functional language using a DSEL. Since our host language has a sufficiently strong type system, we achieve soundness *without needing to construct specialised correctness proofs*. (Section 6)
3. We design, implement and consequently prove correct a concurrent programming notation which guarantees that resources are requested before use and released before exit, and that deadlock does not occur. (Sections 4–6)

Additionally, we introduce a dependently typed programming language, IDRIS¹. This language is intended as a platform for experimentation with practical aspects of dependently typed functional programming such as the interaction with I/O, shared external state and concurrency. To this end, we implement a meta-operation, EXECUTE [11], external to the type theory, which runs *effectful* programs (that is programs that may affect the program environment through I/O or other effects). The code and examples in this paper are directly generated from executable IDRIS programs, available from <http://www.cs.st-and.ac.uk/~eb/ConcDSEL/>.

2. Motivating Example: Safe Concurrent Access to Bank Accounts

Concurrent software is difficult to implement correctly, and difficult to prove correct because the order of evaluation between processes

¹<http://www.cs.st-and.ac.uk/~eb/Idris/>

is difficult or impossible to determine. There are potential difficulties even with simple code. For example, there are (at least) two problems with the following pseudo-code, that attempts to move a sum of money from one bank account to another in a concurrent system:

```
moveMoney(sum, sender, receiver) {
  lock(sender);
  lock(receiver);
  sendFunds = read(sender);
  recvFunds = read(receiver);
  if (sendFunds < sum) {
    putStrLn("Insufficient funds");
    return;
  }
  write(sender, sendFunds - sum);
  write(receiver, recvFunds + sum);
  unlock(receiver);
  unlock(sender);
}
```

This code locks the resource associated with each account, to prevent other processes accessing the account in the critical section between the read and the write. It then moves the money and unlocks the resources.

The first problem occurs if there is insufficient funds in the sender's account. The function reports an error and then exits immediately, forgetting to unlock the resources. This has disastrous consequences, because now no other process can ever access either account! The second problem is harder to spot: imagine the following calls are made simultaneously in separate processes:

1. `moveMoney(20, Simon, Phil)`
2. `moveMoney(10, Phil, John)`
3. `moveMoney(42, John, Simon)`

If each process executes one statement in turn, execution proceeds as follows:

1. Simon sends Phil £20, beginning by executing `lock(Simon)`.
2. Phil sends John £10, beginning by executing `lock(Phil)`.
3. John sends Simon £42, beginning by executing `lock(John)`.
4. Now all three resources are locked — none of the processes can lock the resource associated with the receiver! The system is deadlocked.

This second problem occurs not in the code, but in the execution environment. Nevertheless, the problem arises because of the code — no thought is made as to how to prevent deadlock such as this occurring. A possible solution is to impose a priority ordering on resources (for example, account number) and always lock the higher priority resource first.

Concurrent systems such as this are inherently stateful, and their correctness relies on it being impossible to enter invalid states such as attempting to return from a process without releasing resources, or requesting resources in the wrong order. In traditional imperative programming languages such properties cannot be checked mechanically — correctness relies either on informal coding conventions, a runtime monitor or dynamic checking, or in cases where safety is important, model checking. Even a formal approach such as model checking is not infallible, however — the state space may be large, the system must be correctly transcribed from the model, and the resulting system cannot be modified without reconstructing and re-verifying the model.

In this paper, we will explore how dependently typed functional programming can provide a means for checking such properties

statically and within the implementation itself, thus guaranteeing that resources are requested and released as necessary without causing deadlock. We will do so by implementing a domain specific embedded language (DSEL) within a host dependently typed language. As an example, the program above would be written in our DSEL as follows:

```
moveMoney : Int →
            (sender : Fin n) → (receiver : Fin n) →
            Lang (accounts n) (accounts n) TyUnit
moveMoney sum sender receiver
  ↳ do let sendEl = elemIs sender -
        let recvEl = elemIs receiver -
        LOCK sendEl □1
        LOCK recvEl □2
        sendFunds ← READ sendEl
        recvFunds ← READ recvEl
        CHECK (isLT sendFunds sum)
            (ACTION (putStrLn "Insufficient funds"))
            (λ p. do WRITE sendEl (sendFunds - sum)
                    WRITE recvEl (recvFunds + sum)
                    UNLOCK sendEl
                    UNLOCK recvEl)
```

Unlike the pseudo-code version initially given, our dependent type system will reject this program, firstly because some branches do not unlock resources, and secondly because, as discussed, there is no guarantee that locking the resources in the given order will not cause deadlock. In the rest of this paper, we will see how this DSEL is constructed, and how it can be used to write a *correct* implementation of `moveMoney` and similar programs.

3. Dependently Typed Programming

IDRIS is a full-spectrum dependently typed programming language, similar to EPIGRAM [22] or AGDA², and built on top of the IVOR [4] theorem proving library. It is a pure functional language with a syntax similar to Haskell with GADTs. Functions are *total*, which is guaranteed by requiring that all cases are covered and that recursive calls are on structurally smaller values of strictly positive data types.

The purpose of the language is to provide a platform for practical programming with dependent types. We have used our own implementation, rather than an existing tool, as this gives complete freedom to experiment with abstractions and language features beyond the type system, such as I/O and concurrency. Additionally, although unrelated to the work we present in this paper, the core language is intended as an important step towards a fully dependently typed implementation of Hume [10].

In this section, we introduce some features of the language and some important techniques in dependently typed programming. The reader who is already familiar with dependently typed programming may wish to skip this section.

3.1 Informative Testing

We illustrate the syntax of IDRIS with a simple example showing one of the key techniques in dependently typed programming, namely the use of indexed data types for informative testing.

In a simply-typed programming language, testing a value is completely dynamic. For example, if we were to lookup an index in a list in Haskell, we would use the `!!` function:

```
(!!) :: [a] -> Int -> a
(!!) 0 (x:xs) = x
(!!) n (x:xs) = xs!!(n-1)
```

²<http://agda.sourceforge.net/>

Suppose we use the function as follows:

```
if xs!!i == y then foo xs else bar xs
```

Since we have made a choice based on the i th element of xs , we ought to know *statically* that, in the `then` branch, it is safe to make certain assumptions, e.g. that xs has at least i elements, that y is a member of xs , and so on. As programmers, we regularly make such assumptions based on our own knowledge of the code. The compiler, however, has no such knowledge and cannot therefore guarantee that the assumptions we have made are valid, and we find out about our mistakes only when we encounter a run-time error. In safety critical real-time systems, this may be too late. Dependent types allow us to state and enforce our assumptions. For example, consider the following data type representing vectors (sized lists):

```
data Vect : (A :  $\star$ )  $\rightarrow$  (n :  $\mathbb{N}$ )  $\rightarrow$   $\star$    where
  nil : Vect A 0
  | cons : A  $\rightarrow$  Vect A k  $\rightarrow$  Vect A (s k)
```

Correspondingly, we have the finite sets which we can use to represent bounded numbers:

```
data Fin : (n :  $\mathbb{N}$ )  $\rightarrow$   $\star$    where
  f0 : Fin (s k)
  | fs : Fin k  $\rightarrow$  Fin (s k)
```

We now write the bounds-safe lookup corresponding to `!!`:

```
vlookup : Fin k  $\rightarrow$  Vect A k  $\rightarrow$  A
vlookup f0 (cons x xs)  $\mapsto$  x
vlookup (fs k) (cons x xs)  $\mapsto$  vlookup k xs
```

Since we have dependent types we can do even better. If we know that the i th element of a list xs has the value a , we can write a predicate to state this. The `ElemIs` data type expresses this:

```
data ElemIs : Fin n  $\rightarrow$  A  $\rightarrow$  Vect A n  $\rightarrow$   $\star$    where
  first : ElemIs f0 x (cons x xs)
  | later : ElemIs i x ys  $\rightarrow$  ElemIs (fs i) x (cons y ys)
```

So instead of simply looking up a value, given a list and an index, we compute a predicate which is a proof that the element we looked up is stored at the given index.

```
elemIs : (i : Fin n)  $\rightarrow$  (xs : Vect A n)  $\rightarrow$ 
  ElemIs i (vlookup i xs) xs
elemIs f0 (cons x xs)  $\mapsto$  first
elemIs (fs k) (cons x xs)  $\mapsto$  later (elemIs k xs)
```

Depending on our application, we can use as much or as little of this information as we need. Often, simply looking up the element with `vlookup` is sufficient. However, depending on the nature of the static safety required, we may need to keep the membership proof for longer. In this paper, we will see examples where computing and retaining such proofs gives strong safety guarantees.

3.1.1 Collapsible data structures

`ElemIs` is an example of a **collapsible** data structure [7]. Collapsible data structures have the property that their values can be uniquely determined from their indices. In other words, `ElemIs i x xs` has at most one element for any i , x and xs . Concretely, a type is collapsible if one index has mutually exclusive values for each constructor, and all constructor arguments are recursive. In the case of `ElemIs`, `first` has an index of `f0` and `later` has an index of `fs i`, so the indices uniquely determine the constructor. In addition, the only argument to `later` is recursive. Collapsibility has two important consequences:

- At compile-time, if the indices are statically known, for example from a function type, the value can be filled in automatically.

This is important from a programming point of view, because it means proofs need not be given explicitly.

- At run-time, the value can be discarded as detailed in [7, 3], since it contains no more information than the indices. This is important from an efficiency point of view — we are able to use the computational information that `ElemIs` gives us, without the run-time overhead of storing the proofs.

3.2 I/O with dependent types

We require input/output operations to be executable from within our language. To achieve this, we use Hancock and Setzer's I/O model [11]. This involves implementing a Haskell-style I/O monad [25] 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 implemented in terms of these of commands and responses:

```
data IO :  $\star$   $\rightarrow$   $\star$    where
  IOReturn : A  $\rightarrow$  IO A
  | IODo : (c : Command)  $\rightarrow$ 
    (Response c  $\rightarrow$  IO A)  $\rightarrow$  IO A
```

Here, `IODo` 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:

```
data Command :  $\star$    where
  PutStr : String  $\rightarrow$  Command
  | GetStr : Command
  | ...
```

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.

```
Response : Command  $\rightarrow$   $\star$ 
Response (PutStr s)  $\mapsto$  ()
Response GetStr  $\mapsto$  String
...
```

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

```
getStr : IO String
getStr = IODo GetStr ( $\lambda s$ :String. Return s)
putStr : String  $\rightarrow$  IO Unit
putStr s = IODo (PutStr s) ( $\lambda x$ :Unit. 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 will use Haskell-style `do` notation, with the obvious translation into our `IODo` and `Return` operations):

```
EXECUTE (IODo c r)  $\implies$  RUN c r
EXECUTE (Return v)  $\implies$  return v
RUN PutStr k  $\implies$  do str  $\leftarrow$  getLine
  EXECUTE (k str)
RUN (GetStr s) k  $\implies$  do putStr s
  EXECUTE (k unit)
...
```

For the program we will implement in this paper, we require the following operations, implemented similarly to `getStr` and `putStr` above:

```

fork : IO () → IO ()
newLock : Int → IO Lock
lock, unlock : Lock → IO ()
newIORef : A → IO (IORef A)
readIORef : IORef A → IO A
writeIORef : IORef A → A → IO ()

```

Lock is an externally implemented semaphore, created by `newLock`, with `lock` and `unlock` requesting and releasing the semaphore respectively. The other functions behave in the same way as their Haskell equivalents.

4. A Domain Specific Embedded Language for Concurrent Programming

We would like to use the power of dependent types to verify the correct management of low-level resources. The difficulty is that resource management is inherently stateful, and our meta-language, having full dependent types, is necessarily pure. Additionally we do not wish to restrict any other operations which have no impact on the resource management. To achieve this, we encapsulate the resource and state management operations in a domain specific embedded language (DSEL). In this section, we will consider the features and required properties of the DSEL, and present a simple type system which captures these required properties. We will consider the implementation details in section 5, but in considering our design we keep in mind that a likely implementation technique is via a state monad.

It is important to maintain a clear distinction between the implementation language and the language being implemented. We refer to the implementation language IDRIS as the **meta-language** and the language being implemented, the DSEL, as the **object language**.

4.1 Ensuring Safe Resource Access

One of the key requirements of concurrent programs is that they manage external, shared resources safely. A classic problem which may arise in this setting involves two threads incrementing a shared variable, `var`. Each thread will read the contents of `var`, increment it, and write the result back:

```

increment = do val ← READ var
            WRITE (val + 1) var

```

Clearly, the correct outcome is for value stored in the shared variable to have increased by two. If however, both threads read the variable simultaneously then each thread will increment the original value and the result will be that the value has increased by only one. Naturally, the solution to this race condition is to treat the variable as a shared resource which should be locked throughout the critical sequence. We aim to prevent such errors by requiring that all shared variables are locked before access.

4.2 Preventing Deadlock

A second challenge in more complex concurrent programs is to avoid deadlock. A deadlock occurs when two or more threads are waiting for each other to release a resource. There are four necessary conditions for a deadlock to occur [8]:

1. Resources are mutually exclusive, i.e. only one process can have access at a time.
2. Processes may request the use of additional resources.
3. Only a process with access to a resource may release it.
4. Two or more processes form a circular chain, with each process waiting for a resource currently held by the next process in the chain.

If we can prevent any of these conditions from occurring, then deadlock can never occur. Clearly, the first three will apply in any concurrent system which requires access to shared resources. In our DSEL, we prevent deadlock by enforcing an *ordering* on resources, eliminating condition 4. We prevent circular chains forming by requiring that each resource a process requests has a *lower* priority than those it already holds. While this is slightly restrictive, determining whether deadlock occurs is in general undecidable so any solution we choose which *guarantees* deadlock freedom will necessarily be restrictive. Although there is not always an obvious ordering and it may not always be known in advance which resources a process requires, this method proves useful in practice [29].

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 we take in this paper is to use dependent types to statically enforce the required constraints on the usage of resources such as locks. In this way, we can prevent the *construction* of programs that could violate these constraints, by ensuring, for example, that race conditions such as described above can never be encountered.

4.3 Syntax and Semantics

The abstract syntax of our DSEL is given in Figure 1. At appropriate points, we allow embedding of terms in the meta-language. The type structure is fairly simple — there is a unit type `Unit` and a means of lifting meta-language types into the object language, `Lift`. The language is defined relative to a set of resources. Resources are shared variables, associated with semaphores. We have a context Δ which keeps track of the type and lock status of the resource.

$$\begin{aligned}
T &::= \text{Unit} \mid \text{Lift } t & x &::= \langle \text{variable} \rangle & r &::= \langle \text{resource id} \rangle \\
s &::= \text{LOCK } r \mid \text{UNLOCK } r \mid \text{READ } r \mid \text{WRITE } t r \mid \\
&\quad \text{LOOP } n p \mid \text{FORK } p \mid \text{RETURN } t \mid \\
&\quad \text{ACTION } t \mid \text{CHECK } t p p \\
p &::= s \mid x \leftarrow p; p \mid p; p & t &::= \text{meta-language term}
\end{aligned}$$

Figure 1. Syntax of the Concurrent DSEL

The language includes several control constructs; `LOOP` which executes the body n times, `CHECK` which branches according to a value of type `Maybe`, and, most importantly for a concurrent language, `FORK` which spawns a new thread. We can also lift meta-language values into the object language with `RETURN` and execute an arbitrary I/O action with `ACTION`. These actions are potentially useful for any state managing DSEL.

The language also includes operations specific to the concurrency domain. `LOCK` and `UNLOCK` request and release a resource respectively. `READ` reads a value from a resource, and `WRITE` writes a value back to a resource. These are the operations which we must constrain in order to guarantee freedom from deadlock and invalid resource accesses. The typing rules which enforce these constraints are given in figure 2.

The typing rules are defined relative to the context Δ , in order to know the type of values which can be read and written. They are also defined relative to the meta-language's context Γ , since it is meta-language values which are stored in the shared variables. There are therefore two typing judgments:

- $\Gamma \vdash a : T$, the term a has the type T relative to context Γ in the meta-language.

- $\Gamma, \Delta \vdash p : T$, the program p has the type T relative to the context Γ in the meta-language and the resource mapping Δ in the object language.

$$\begin{array}{c}
\frac{\Delta \vdash \text{valid} \quad r_{\text{resource}}}{\Gamma, \Delta \vdash \text{LOCK } r : \text{Unit}} [\text{no lower priority resources locked}] \\
\frac{\Delta \vdash \text{valid} \quad r_{\text{resource}}}{\Gamma, \Delta \vdash \text{UNLOCK } r : \text{Unit}} [x \text{ locked at least once}] \\
\frac{\Delta \vdash r \mapsto T}{\Gamma, \Delta \vdash \text{READ } r : \text{Lift } T} [x \text{ locked}] \\
\frac{\Delta \vdash r \mapsto T \quad \Gamma \vdash v : T}{\Gamma, \Delta \vdash \text{WRITE } v x : \text{Unit}} [x \text{ locked}] \\
\frac{\Gamma, \Delta \vdash p : \text{Unit}}{\Gamma, \Delta \vdash \text{FORK } p : \text{Unit}} \quad \frac{\Gamma \vdash v : T}{\Gamma, \Delta \vdash \text{RETURN } v : \text{Lift } T} \\
\frac{\Gamma \vdash v : \text{Maybe } A \quad \Gamma, \Delta \vdash n : S \quad \Gamma; v : A, \Delta \vdash j : S}{\Gamma, \Delta \vdash \text{CHECK } v n j : S} \\
\frac{\Gamma \vdash n : \mathbb{N} \quad \Gamma, \Delta \vdash p : \text{Unit}}{\Gamma, \Delta \vdash \text{LOOP } n p : \text{Unit}} \quad \frac{\Gamma \vdash v : \text{IO } ()}{\Gamma, \Delta \vdash \text{ACTION } v : \text{Unit}} \\
\frac{\Gamma, \Delta \vdash p : T \quad \Gamma; x : T, \Delta \vdash p' : T' \quad \Delta}{\Gamma, \Delta \vdash x \leftarrow p; p' : T}
\end{array}$$

Figure 2. Typing Rules for the Concurrent DSEL

An advantage of our approach is that the constraints on the typing rules can remain informal, as no soundness proofs are required. This is, we need not express here precisely *how* we establish that no lower priority resources are locked in the rule for LOCK. Such details can be deferred to the implementation, since we are able to express such properties precisely in the implementation. Being able to defer such formal details to the implementation means that we can concentrate on the intuition behind the rules and ensure that the meaning is correct. Since IDRIS is built on a sound dependent type system, we can be sure our typing rules and their implementation are sound (that is, they have the usual properties of type preservation under reduction, Church-Rosser, etc), provided that these constraints are expressed correctly.

5. State Handling DSELS

In this section we consider how to implement a state handling DSEL in IDRIS. We begin considering the general form of such a language implementation, before specialising it to the concurrency domain. In general, the programs we consider will:

- Manipulate state
- Compute a value

Our goal is static safety, therefore as far as possible we will try to reflect this behaviour in our representations. The type of the language representation in particular should reflect an input state, an output state and a return value.

In our initial implementation, we will not consider what form the states might take, nor any domain-specific types. We limit the initial presentation to control constructs (i.e variable binding and the LOOP, FORK, CHECK, ACTION and RETURN commands).

5.1 Types

An expression in the language may execute some action (returning the unit type) or return a type in the meta-language. Our representation of types reflects this:

$$\begin{array}{l}
\text{data } \text{Ty} : \star \quad \text{where} \\
\quad \text{TyUnit} : \text{Ty} \\
\quad | \quad \text{TyLift} : \star \rightarrow \text{Ty}
\end{array}$$

Types can be converted to meta-language types in the obvious way:

$$\begin{array}{l}
\text{interpTy} : \text{Ty} \rightarrow \star \\
\text{interpTy } \text{TyUnit} \mapsto () \\
\text{interpTy } (\text{TyLift } A) \mapsto A
\end{array}$$

This representation gives complete flexibility in the choice of domain specific types. We treat the unit type specially (rather than simply representing all types as meta-language types) because it fulfills a special rôle in the object language, and additionally maintains a clear logical separation between meta-language types (of type \star) and object language types (of type Ty).

5.2 Language Representation

The language modifies state, and returns a value. Our representation should therefore reflect this:

$$\text{data } \text{Lang} : \text{StateIn} \rightarrow \text{StateOut} \rightarrow \text{Ty} \rightarrow \star$$

We leave the definitions of *StateIn* and *StateOut* for the moment, as these will depend on the specific domain. For complete flexibility, we allow the input and output state to be different types (this may help in particular if an operation adds a new resource to a state modelled as a *Vect*, since modifying the index of a *Vect* is modifying the type.). *StateIn* and *StateOut* correspond to the Δ in the typing rules in Figure 2. Figure 3 gives the declaration of the common DSEL constructs.

$$\begin{array}{l}
\text{data } \text{Lang} : \text{StateIn} \rightarrow \text{StateOut} \rightarrow \text{Ty} \rightarrow \star \quad \text{where} \\
\text{LOOP} : (\text{count} : \text{Nat}) \rightarrow \\
\quad (\text{body} : \text{Lang } s_{\text{in}} s_{\text{in}} \text{TyUnit}) \rightarrow \\
\quad \text{Lang } s_{\text{in}} s_{\text{in}} \text{TyUnit} \\
| \text{FORK} : (\text{proc} : \text{Lang } s_{\text{in}} s_{\text{in}} \text{TyUnit}) \rightarrow \\
\quad \text{Lang } s_{\text{in}} s_{\text{in}} \text{TyUnit} \\
| \text{CHECK} : \text{Maybe } a \rightarrow \\
\quad (\text{ifJ} : a \rightarrow \text{Lang } s_{\text{in}} s_{\text{out}} \text{ty}) \rightarrow \\
\quad (\text{ifN} : \text{Lang } s_{\text{in}} s_{\text{out}} \text{ty}) \rightarrow \\
\quad \text{Lang } s_{\text{in}} s_{\text{out}} \text{ty} \\
| \text{ACTION} : \text{IO } () \rightarrow \text{Lang } s_{\text{in}} s_{\text{in}} \text{TyUnit} \\
| \text{RETURN} : (\text{val} : \text{interpTy } \text{ty}) \rightarrow \text{Lang } s_{\text{in}} s_{\text{in}} \text{ty} \\
| \text{BIND} : (\text{code} : \text{Lang } s_{\text{in}} s_{\text{k}} \text{ty}) \rightarrow \\
\quad (\text{k} : \text{interpTy } \text{ty} \rightarrow \text{Lang } s_{\text{k}} s_{\text{out}} \text{tyout}) \rightarrow \\
\quad \text{Lang } s_{\text{in}} s_{\text{out}} \text{tyout}
\end{array}$$

Figure 3. Common DSEL constructs

The principal advantage of embedding a domain specific language in a host language is that it allows us to exploit features of the host language. Choosing a dependently typed language as the host language means that we can exploit the host type system to represent the *typing rules* of the object language from Figure 2.

Each constructor of the *Lang* datatype corresponds to the syntactic constructs in the DSEL. Since the language is parametrised over types, this also means that we can express the language's typing rules in the constructors. e.g. RETURN, given a value in the meta-language, constructs a value in the object language. The corresponding constructor in the *Lang* type reflects this:

$$\text{RETURN} : (\text{val} : \text{interpTy } \text{ty}) \rightarrow \text{Lang } s_{\text{in}} s_{\text{in}} \text{ty}$$

A second, equally important, feature of the host language we exploit is variable binding. To bind a value, we compute an expression and continue execution with the context Γ extended with

that value. It is not surprising, therefore, to find that the type of our binding construct is similar to an ordinary monadic bind, with the addition of the explicit threading of state:

$$\text{BIND} : (\text{code} : \text{Lang } s_{in} s_k ty) \rightarrow \\ (k : \text{interpTy } ty \rightarrow \text{Lang } s_k s_{out} tyout) \rightarrow \\ \text{Lang } s_{in} s_{out} tyout$$

Even though our goal is total static safety, dynamic checking is regularly required in any program. For this purpose, we include the CHECK construct. This takes the result of a computation in the meta-language, of type Maybe a , and continues execution according to whether the value of type a exists. The a might typically be a dynamically computed proof — in the situation where a proof is not statically known, we can at least guarantee that a program has carried out necessary dynamic checks.

$$\text{CHECK} : \text{Maybe } a \rightarrow \\ (ifJ : a \rightarrow \text{Lang } s_{in} s_{out} ty) \rightarrow \\ (ifN : \text{Lang } s_{in} s_{out} ty) \rightarrow \\ \text{Lang } s_{in} s_{out} ty$$

The representation so far gives us the ability to lift meta-language values into the object language, bind values, and control program flow. These are important components of many state managing domain specific languages. However, until we define *StateIn* and *StateOut* and operations to manipulate them, we will not be able to do anything interesting.

6. A DSEL For Concurrent Programming

In this section we complete our DSEL for concurrent programming by defining the state over which the language is parametrised and the operations which affect the state. These remaining operations are READ, WRITE, LOCK and UNLOCK. The typing rules for each have informally stated side conditions. To represent these accurately, we must state express the side conditions formally, which we will do directly in code.

6.1 Resources

Each side condition refers to the lock state of a resource. In this state, we need to represent the number of times each resource has been locked, as well as the type of the value associated with the resource:

$$\text{data ResState} : \star \text{ where} \\ \text{RState} : \mathbb{N} \rightarrow \text{Ty} \rightarrow \text{ResState}$$

Additionally, when we run the program, we will need access to the concrete data associated with the resource. This is the semaphore controlling access to the resource, and the value itself. Since a value is mutable external state, we store it in an IORef. The Resource type is parametrised over its state, meaning that we are always able to link a resource, which exists only at run-time, with its state which is known at compile-time.

$$\text{data Resource} : \text{ResState} \rightarrow \star \text{ where} \\ \text{resource} : \text{IORef } (\text{interpTy } a) \rightarrow \text{Lock} \rightarrow \\ \text{Resource } (\text{RState } n a)$$

Programs in our DSEL exist relative to a collection of statically known resources. It is convenient to represent this collection as a Vect, then resources are referred to by an index into this Vect, stored as a Fin n where n is the number of resources. We can now define *StateIn* and *StateOut* in concrete terms:

$$\text{data Lang} : \text{Vect ResState } tin \rightarrow \text{Vect ResState } tout \rightarrow \\ \text{Ty} \rightarrow \star$$

This type declaration, with differing lengths on the input and output state vectors, gives us the potential to create or destroy

resources within a DSEL program. For the moment, however, we assume that the resources are known in advance. This restriction turns out to be minor, because the language is embedded in a host language. While resources cannot be created *within* the DSEL, they can be freely constructed, dynamically, before being passed to a DSEL program.

6.2 Resource Validity

To prevent deadlock, we need to prove that resources are locked in priority order. Storing resources as a vector gives a convenient ordering, namely position in the list. We treat resources which appear earlier in the list as lower priority. Therefore, whenever we attempt to lock a resource, we must be certain that no lower priority resource has already been locked. The predicate PriOK can be constructed if everything before a given index in a vector is an unlocked resource.

$$\text{data PriOK} : (i : \text{Fin } n) \rightarrow \\ (xs : \text{Vect ResState } n) \rightarrow \star \text{ where} \\ \text{isFirst} : \text{PriOK } f0 (\text{cons } x xs) \\ | \text{isLater} : \text{PriOK } i xs \rightarrow \\ \text{PriOK } (fs i) (\text{cons } (\text{RState } 0 t) xs)$$

It is worth considering briefly how we arrive at this definition. Consider first how we might check this dynamically, returning a truth value. The first resource is always safe to lock, since it is lowest priority. A resource in the tail of a list can only be locked if the resource at the head (i.e., the lowest priority resource) is unlocked:

$$\text{isPriOK} : \text{Fin } n \rightarrow \text{Vect ResState } n \rightarrow \text{Bool} \\ \text{isPriOK } f0 (\text{cons } x xs) \mapsto \text{True} \\ \text{isPriOK } (fs i) (\text{cons } (\text{RState } 0 t) xs) \mapsto \text{isPriOK } i xs \\ \text{isPriOK } - - \mapsto \text{False}$$

The branch which returns True corresponds to the isFirst constructor, and the branch which makes the recursive call corresponds to the isLater constructor. The patterns correspond to the indices in PriOK. There is no constructor for the branch which returns False. So, arriving at the definition of such a predicate corresponds directly to implementing the relevant dynamic check. In fact, as discussed in section 3.1, we ought to make this test more informative:

$$\text{isPriOK} : (i : \text{Fin } n) \rightarrow (xs : \text{Vect ResState } n) \rightarrow \\ \text{Maybe } (\text{PriOK } i xs) \\ \text{isPriOK } f0 (\text{cons } x xs) \mapsto \text{Just isFirst} \\ \text{isPriOK } (fs i) (\text{cons } (\text{RState } 0 t) xs) \\ \mapsto \text{mMap isLater } (\text{isPriOK } i xs) \\ \text{isPriOK } - - \mapsto \text{Nothing}$$

Where we previously returned False, we now return Nothing. Instead of returning True we now not only say that the priority is okay, we also *explain why*. We use mMap to lift isLater into the Maybe type.

We will also often need static knowledge of a resource state. To do this we use the ElemIs predicate, defined in section 3.1, to express that we must know the state of a particular resource before we can proceed.

6.3 The Full Language

We are now in position to add the domain specific constructors to Lang for the LOCK, UNLOCK, READ and WRITE operations.

To lock a resource, we must know that it is safe to lock in the current state. We add the following constructor to Lang:

$$\text{LOCK} : (\text{locked} : \text{ElemIs } i (\text{RState } k ty) tins) \rightarrow \\ (\text{priOK} : \text{PriOK } i tins) \rightarrow \\ \text{Lang } tins (\text{update } i (\text{RState } (s k) ty) tins) \text{ TyUnit}$$

This type expresses that locking a value modifies the state, incrementing the lock count on resource i . The **update** function simply updates the value at a position in a **Vect**, and it is convenient simply to be able to lift this into the type of **LOCK**:

$$\mathbf{update} : (i : \text{Fin } n) \rightarrow A \rightarrow \text{Vect } A \ n \rightarrow \text{Vect } A \ n$$

Additionally, requesting a lock is possible only if there are no lower priority resources than i locked. This was the side condition given informally in Figure 2, and is now given formally as an instance of **PriOK**. We can arrive at a proof of this either through static knowledge (e.g., we know at compile-time that no other resources have been locked by this process), or by a dynamic check with **CHECK** and an application of **isPriOK**.

Unlocking a resource makes sense only if the resource has been locked at least once. Again it updates the state, this time by reducing the lock count:

$$\mathbf{UNLOCK} : (\text{locked} : \text{Elem} i \ (\text{RState } (s \ k) \ ty) \ tins) \rightarrow \text{Lang } tins \ (\mathbf{update} \ i \ (\text{RState } k \ ty) \ tins) \ \text{TyUnit}$$

The purpose of locking a resource is, of course, to prevent accessing and modifying those resources in concurrently executing processes. Therefore, we allow reading from and writing to a shared variable only when we can prove that the resource which protects it is locked:

$$\begin{aligned} \mathbf{READ} & : (\text{locked} : \text{Elem} i \ (\text{RState } (s \ k) \ ty) \ tins) \rightarrow \text{Lang } tins \ tins \ ty \\ \mathbf{WRITE} & : (\text{val} : \mathbf{interpTy} \ ty) \rightarrow (\text{locked} : \text{Elem} i \ (\text{RState } (s \ k) \ ty) \ tins) \rightarrow \text{Lang } tins \ tins \ \text{TyUnit} \end{aligned}$$

Neither of these modifies the resource state — the state carries only the number of times a resource is locked and the type of the variable. However, neither is valid unless there is a proof that the resource has previously been locked. Again, such proofs can be constructed dynamically or statically as necessary.

Figure 4 gives the complete language representation.

6.4 Executing the DSEL

Of course, the language is no use without a means of executing programs. Fortunately, implementing the interpreter is largely straightforward and consists of executing the relevant low level operations for each command.

6.4.1 Environments

The interpreter is defined relative to an environment, which carries the concrete values associated with each resource. Environments can be defined generically as mappings from a vector of some type to a vector of interpretations of that type:

$$\begin{aligned} \mathbf{data} \quad \mathbf{Env} & : (R : \star) \rightarrow (iR : R \rightarrow \star) \rightarrow (xs : \text{Vect } R \ n) \rightarrow \star \quad \mathbf{where} \\ \mathbf{Empty} & : \mathbf{Env} \ R \ iR \ \text{nil} \\ | \ \mathbf{Extend} & : (res : iR \ r) \rightarrow \mathbf{Env} \ R \ iR \ xs \rightarrow \mathbf{Env} \ R \ iR \ (\text{cons } r \ xs) \end{aligned}$$

Looking up a value in an environment corresponds to looking up a value in the vector of types:

$$\begin{aligned} \mathbf{envLookup} & : (i : \text{Fin } n) \rightarrow \mathbf{Env} \ R \ iR \ xs \rightarrow iR \ (\mathbf{vlookup} \ i \ xs) \\ \mathbf{envLookup} \ f0 \ (\mathbf{Extend} \ t \ env) & \mapsto t \\ \mathbf{envLookup} \ (fs \ i) \ (\mathbf{Extend} \ t \ env) & \mapsto \mathbf{envLookup} \ i \ env \end{aligned}$$

In our DSEL, we have a vector of **ResState**, where the interpretation is **Resource**. We define resource environments as follows:

$$\begin{aligned} \mathbf{REnv} & : (xs : \text{Vect } \text{ResState} \ n) \rightarrow \star \\ \mathbf{REnv} \ xs & \mapsto \mathbf{Env} \ \text{ResState} \ \text{Resource} \ xs \end{aligned}$$

At various points, we will need to extract references and resource locks from the environment. To lookup a resource lock, with **lookup**, we simply take an index into the environment. To get the reference, with **rlookup**, we also take the **Elem**s proof that a value exists at the index. While this is not strictly necessary, it helps typechecking applications of **rlookup** if the type of the resource is named. Specifically, in the case for **READ_p**, it is much simpler to be able to pass the proof p directly and return a value of the appropriate type given in the proof.

$$\begin{aligned} \mathbf{lookup} & : (i : \text{Fin } n) \rightarrow \mathbf{REnv} \ xs \rightarrow \text{Lock} \\ \mathbf{rlookup} & : (p : \text{Elem} i \ (\text{RState } k \ ty) \ xs) \rightarrow \mathbf{REnv} \ xs \rightarrow \text{IORef} \ (\mathbf{interpTy} \ ty) \end{aligned}$$

6.4.2 The Interpreter

The interpreter takes an environment, and returns a pair of the new environment (since programs modify state) and the result of program execution (since programs are typed).

$$\mathbf{interp} : \mathbf{REnv} \ ty_{in} \rightarrow \text{Lang } ty_{in} \ ty_{out} \ T \rightarrow \text{IO} \ (\text{Pair} \ (\mathbf{REnv} \ ty_{out}) \ (\mathbf{interpTy} \ T))$$

The type expresses that the interpreter must modify the environment to correspond exactly with modifications, if any, to the resource state in the program. Additionally, if a program returns a type T , the interpreter must return the meta-language interpretation of T . Implementation of the interpreter is straightforward — the complete definition is given in Figure 5.

$$\begin{aligned} \mathbf{interp} & : \mathbf{REnv} \ ty_{in} \rightarrow \text{Lang } ty_{in} \ ty_{out} \ T \rightarrow \text{IO} \ (\text{Pair} \ (\mathbf{REnv} \ ty_{out}) \ (\mathbf{interpTy} \ T)) \\ \mathbf{interp} \ env \ (\mathbf{READ} \ p) & \mapsto \mathbf{do} \ \text{val} \ \leftarrow \ \mathbf{readIORef} \ (\mathbf{rlookup} \ p \ env) \\ & \quad \mathbf{return} \ (\text{MkPair} \ env \ \text{val}) \\ \mathbf{interp} \ env \ (\mathbf{WRITE} \ v \ p) & \mapsto \mathbf{do} \ \mathbf{writeIORef} \ (\mathbf{rlookup} \ p \ env) \ v \\ & \quad \mathbf{return} \ (\text{MkPair} \ env \ ()) \\ \mathbf{interp} \ env \ (\mathbf{LOCK} \ p \ pri) & \mapsto \mathbf{do} \ \mathbf{lock} \ (\mathbf{lookup} \ i \ env) \\ & \quad \mathbf{return} \ (\text{MkPair} \ (\mathbf{lockEnv} \ p \ env) \ ()) \\ \mathbf{interp} \ env \ (\mathbf{UNLOCK} \ p) & \mapsto \mathbf{do} \ \mathbf{unlock} \ (\mathbf{lookup} \ i \ env) \\ & \quad \mathbf{return} \ (\text{MkPair} \ (\mathbf{unlockEnv} \ p \ env) \ ()) \\ \mathbf{interp} \ env \ (\mathbf{ACTION} \ io) & \mapsto \mathbf{do} \ io \\ & \quad \mathbf{return} \ (\text{MkPair} \ env \ ()) \\ \mathbf{interp} \ env \ (\mathbf{RETURN} \ val) & \mapsto \mathbf{return} \ (\text{MkPair} \ env \ val) \\ \mathbf{interp} \ env \ (\mathbf{FORK} \ proc) & \mapsto \mathbf{do} \ \mathbf{fork} \ (\mathbf{do} \ f \ \leftarrow \ \mathbf{interp} \ env \ proc \\ & \quad \mathbf{return} \ ()) \\ & \quad \mathbf{return} \ (\text{MkPair} \ env \ ()) \\ \mathbf{interp} \ env \ (\mathbf{BIND} \ code \ k) & \mapsto \mathbf{do} \ (\text{MkPair} \ env' \ val) \ \leftarrow \ \mathbf{interp} \ env \ code \\ & \quad \mathbf{interp} \ env' \ (k \ val) \end{aligned}$$

Figure 5. The DSEL interpreter

6.5 Example

To demonstrate how this language works in practice, we consider a simple example where the resource state is statically known. In this example, we have two shared variables, both \mathbb{N} , and we set up the vector of resource states as follows:

$$\mathbf{nats} = \text{cons} \ (\text{RState} \ 0 \ (\text{TyLift} \ \mathbb{N})) \ (\text{cons} \ (\text{RState} \ 0 \ (\text{TyLift} \ \mathbb{N})) \ \text{nil})$$

```

data Lang : Vect ResState tin → Vect ResState tout → Ty → * where
  READ : (locked : ElemIs i (RState (s k) ty) tins) → Lang tins tins ty
  | WRITE : (val : interpTy ty) → (locked : ElemIs i (RState (s k) ty) tins) → Lang tins tins TyUnit
  | LOCK : (locked : ElemIs i (RState k ty) tins) → (priOK : PriOK i tins) →
          Lang tins (update i (RState (s k) ty) tins) TyUnit
  | UNLOCK : (locked : ElemIs i (RState (s k) ty) tins) → Lang tins (update i (RState k ty) tins) TyUnit
  | LOOP : (count : ℕ) → (body : Lang tins tins TyUnit) → Lang tins tins TyUnit
  | FORK : (proc : Lang tins tins TyUnit) → Lang tins tins TyUnit
  | CHECK : Maybe a → (ifJ : a → Lang tins touts ty) → (ifN : Lang tins touts ty) → Lang tins touts ty
  | ACTION : IO () → Lang tins tins TyUnit
  | RETURN : (val : interpTy ty) → Lang tins tins ty
  | BIND : (code : Lang tins ts1 ty) → (k : interpTy ty → Lang ts1 touts tyout) → Lang tins touts tyout

```

Figure 4. Concurrency DSEL syntax representation

Then we can write a program which loops, incrementing each variable and printing the sum on each iteration. For readability, we have used `do` notation in place of the `BIND` instruction. Although this is not yet implemented, it is a trivial transformation to insert the `BIND` calls in place of the `do` notation:

```

count : ℕ → String → Lang nats nats TyUnit
count n pid
  ↪ LOOP n (do LOCK (later first) □1
            LOCK first □2
            numa ← READ first
            WRITE (S numa) first
            numb ← READ (later first)
            WRITE (S numb) (later first)
            UNLOCK (later first)
            UNLOCK first
            ACTION (putStrLn ((pid++"Val: ")
                               ++show (plus numa numb)))

```

This program is thread-safe — the type ensures that all resources are unlocked on exit, and the language statically ensures that resources are locked in priority order. Since the resources are statically known, the proofs of ordering are easy to construct. `LOCK` and `UNLOCK` in fact refer to resources by proofs of membership in the list, constructed with `first` and `later`.

We refer to locks by their proofs, because `first` is *always* a proof (of type `ElemIs`) that the first resource has the correct number of locks. Given a proof p that the n th resource is correct, `later p` is a proof that the $(n + 1)$ th resource has the correct number of locks. The details of the proof can always be completed by unification if we are using the resource correctly, because `ElemIs` is *collapsible*, as described in section 3.1.1.

We have left two **holes** in this program, \square_1 and \square_2 . Since dependently typed programs often contain proof terms, it can be convenient (and aid readability) to leave gaps for those proof terms. The typechecker will infer the required type for each hole, which we can fill in later. This helps the programmer concentrate on the algorithm without having to keep in mind all of the necessary proof obligations. In this case the obligations are:

```

□1 : PriOK (fs f0) (cons (RState 0 ℕ)
                        (cons (RState 0 ℕ) nil))
□2 : PriOK f0 (cons (RState 0 ℕ)
                    (cons (RState (s 0) ℕ) nil))

```

The first hole requires a proof that there is no lower priority resource than the second locked. This is solved by `isLater isFirst`. The second hole requires a proof that there is no lower priority resource than the first locked under the assumption that the second is locked. This is solved by `first`.

If we try to lock the resources in the wrong order, or do not lock the resources before reading or writing, typechecking will fail because these proofs will not unify with the function type. Since it does typecheck, we can safely execute the program in multiple threads:

```

threadcount : Lang nats nats TyUnit
threadcount ↪ BIND (FORK (count ten "thread")
                    (λu.count ten "main"))

```

Note that in several places this program makes use of the host language. In particular, the `count` function takes two parameters passed using the host language. To execute this program, we need to construct an initial environment with the concrete locks and references:

```

mkNatEnv : IO (REnv nats)
mkNatEnv ↪ do r1 ← newIORef 0
           l1 ← newLock 1
           r2 ← newIORef 0
           l2 ← newLock 1
           return (Extend (resource r1 l1)
                        (Extend (resource r2 l2) Empty))

```

Then we can run the program by calling the interpreter with this initial environment.

```

runThreads : IO ()
runThreads ↪ do env ← mkNatEnv
             p ← interp env threadcount
             return ()

```

Efficiency

There is an apparent abstraction overhead in our approach, by introducing the intermediate interpreter stage, which eventually executes the low level instructions. We do not yet have a compiler for `IDRIS` — our implementation is an interpreter in Haskell — so we do not yet have realistic performance results. However, we believe that a compiled implementation will be competitive with a direct implementation: Since `interp` is a total function, and programs such as `runThread` and `count` above are known at compile-time, the interpreter can be partially evaluated with the input program. Staging an interpreter in this way, resulting in a translator to the host language, is a well-known technique [31, 9] and applies equally well to dependently typed languages [23, 6]

6.6 Correctness

When implementing a new language with a strong type system, especially one with strong safety properties such as our DSEL,

it is important that certain metatheoretic properties are satisfied, and that the actual implementation corresponds to the typing rules. Since our host language is strong enough to express the typing rules directly, it is easy to show that the implementation corresponds to the rules, and many of the required properties follow directly. We do not need separate proofs for these properties because the implementation *is* the proof.

6.6.1 Domain-Specific Properties

Our DSEL satisfies two domain-specific properties: i) resource access is *safe* in that it is impossible to access a resource which is not locked, and ii) deadlock cannot occur. The first is satisfied because all resource accesses (i.e. READs and WRITEs must be accompanied by a proof that they are locked in the current state. The second is satisfied by enforcing a priority ordering on resources which eliminates the possibility of cyclic chains of resource requests. These properties cover all of the constraints in the typing rules.

6.6.2 General Properties

The interpreter for the DSEL inherits properties of the host language by giving it a sufficiently informative type:

$$\text{interp} : \mathbf{REnv} \, ty_{in} \rightarrow \text{Lang} \, ty_{in} \, ty_{out} \, T \rightarrow \text{IO} (\text{Pair} (\mathbf{REnv} \, ty_{out}) (\text{interpTy} \, T))$$

This type expresses the following properties directly, which are verified by the typechecker:

- Interpretation is type preserving — a program with type T will always return a value in the host language interpretation of T .
- Interpretation maintains state according to the typing rules, in that the input and output environments are defined relative to the input and output states. This is important, because the typing rules, especially side conditions on locking and resource access, rely on maintaining state.

A significant advantage of our approach is that these properties follow directly from the *implementation*. If we change or extend either the typing rules or the interpreter, the typechecker ensures that they remain consistent with each other.

6.7 The Next 700 Domain-Specific Embedded Languages?

We have described the implementation in two stages, namely: implementing a generic set of control constructs, and implementing the resource management operations. Since the resource management is independent of the control structures, and we may wish to apply this technique to other resources (such as file handles or even stack and heap usage), it would be desirable to make this separation explicit in the code. An alternative approach, therefore, rather than adding constructors to our initial DSEL, would be to parametrise it over the state handling fragment of the language, and add a constructor which lifts the sublanguage into the main language:

$$\begin{aligned} \text{data} \quad \text{Lang} & : (L : \text{StateIn}' \rightarrow \text{StateOut}' \rightarrow \text{Ty}) \rightarrow \\ & \quad \text{StateIn} \rightarrow \text{StateOut} \rightarrow \\ & \quad \text{Ty} \rightarrow \star \quad \text{where} \\ \text{lift} & : (\text{prog} : L \, s_{in} \, s_{out} \, T) \rightarrow \text{Lang} \, L \, s_{in} \, s_{out} \, T \\ & | \dots \end{aligned}$$

We then define the resource language independently:

$$\begin{aligned} \text{data} \quad \text{RLang} & : \text{Vect ResState} \, tin \rightarrow \text{Vect ResState} \, tout \rightarrow \\ & \quad \text{Ty} \rightarrow \star \quad \text{where} \\ \text{READ} & : (\text{locked} : \text{ElemIs} \, i \, (\text{RState} \, (s \, k) \, ty) \, tins) \rightarrow \\ & \quad \text{RLang} \, tins \, tins \, ty \\ | \quad \text{WRITE} & : (\text{val} : \text{interpTy} \, ty) \rightarrow \\ & \quad (\text{locked} : \text{ElemIs} \, i \, (\text{RState} \, (s \, k) \, ty) \, tins) \rightarrow \\ & \quad \text{RLang} \, tins \, tins \, \text{TyUnit} \\ & \dots \end{aligned}$$

The interpreter for Lang is similarly parametrised over an interpreter for RLang. This follows Landin [18] in that it clearly separates, as Landin puts it, a “basic set of given things” (the resource language) from the “way of expressing things in terms of other things” (the main language). To implement a new state managing DSEL requires only the definition and the interpreter for the sublanguage. Additionally, we anticipate that further development of this approach will allow composition of DSELs from smaller component DSELs.

7. Bank Accounts Revisited

In the example we gave in section 6.5, the resources were statically known. In many cases this will be sufficient, for example in an embedded system managing shared access to specific hardware components. What happens if the resources are not statically known, for example in a database?

We return to the motivating example of section 2 and consider how our DSEL can be used to implement safe concurrent access to bank accounts. Each account is a shared resource, of which there can be an arbitrary number. We will implement the `moveMoney` function, where the accounts concerned will be determined *dynamically*. The account data we hold is simply the amount of money available:

$$\begin{aligned} \text{data} \quad \text{AccountData} & : \star \quad \text{where} \\ \text{MkAcc} & : \text{Int} \rightarrow \text{AccountData} \end{aligned}$$

Each account is associated with a resource, as such each concurrent function should be parametrised over a list of n resource states, where n is decided at run-time.

$$\begin{aligned} \text{accounts} & : (n : \mathbb{N}) \rightarrow \text{Vect ResState} \, n \\ \text{accounts} \, 0 & \mapsto \text{nil} \\ \text{accounts} \, (s \, k) & \mapsto \text{cons} (\text{RState} \, 0 \, (\text{TyLift} \, \text{AccountData})) \\ & \quad (\text{accounts} \, k) \end{aligned}$$

We assume there is a mapping from account numbers and sort codes to a resource identifier (i.e. an index into the vector of accounts):

$$\text{getID} : (\text{sortCode} : \text{Int}) \rightarrow (\text{accountNo} : \text{Int}) \rightarrow \text{Fin} \, n$$

We will initialise the system by constructing a number of accounts, associating each with a lock and an initial sum of money, which, in our somewhat unrealistic world, we distribute equally:

$$\begin{aligned} \text{mkAccounts} & : (n : \mathbb{N}) \rightarrow \text{IO} (\mathbf{REnv} (\text{accounts} \, n)) \\ \text{mkAccounts} \, 0 & \mapsto \text{return Empty} \\ \text{mkAccounts} \, (s \, k) & \\ & \mapsto \text{do} \, kaccs \leftarrow \text{mkAccounts} \, k \\ & \quad aref \leftarrow \text{newIORef} (\text{MkAcc} \, 10000) \\ & \quad alock \leftarrow \text{newLock} \, 1 \\ & \quad \text{return} (\text{Extend} (\text{resource} \, aref \, alock) \, kaccs) \end{aligned}$$

Let us try to implement a function which moves money between accounts.

7.1 First attempt

Recall our first (incorrect) attempt at implementing `moveMoney` in section 2. With our type system, we should therefore expect it to fail. We repeat the attempted definition below:

```

moveMoney : Int →
  (sender : Fin n) → (receiver : Fin n) →
  Lang (accounts n) (accounts n) TyUnit
moveMoney sum sender receiver
  ↪ do let sendEl = elemIs sender -
      let recvEl = elemIs receiver -
      LOCK sendEl □1
      LOCK recvEl □2
      sendFunds ← READ sendEl
      recvFunds ← READ recvEl
      CHECK (isLT sendFunds sum)
      (ACTION (putStrLn "Insufficient funds"))
      (λ p. do WRITE sendEl (sendFunds - sum)
          WRITE recvEl (recvFunds + sum)
          UNLOCK sendEl
          UNLOCK recvEl)

```

(**Aside:** The `let` binding we use here is really a macro, since `elemIs sender` has a different type at each instance despite being syntactically the same. We have used `let` here for readability, but in the actual implementation, we use the expansion.)

The problems identified in section 2 manifest themselves in the following ways:

- After the CHECK, in the branch where there are insufficient funds in the sender's account, there are no UNLOCK instructions. The state on exit here has two *locked* resources, and the type of `moveMoney` requires that the state on exit has all resources unlocked. This is a type error.
- The proof term \square_2 is impossible to fill in, because we cannot guarantee that the sender, being locked first, is always the highest priority resource. \square_1 is fine, because it is always possible to lock a resource when everything is unlocked.

The first problem is easy to fix — we can simply lift the unlock-ing outside the CHECK. To solve the second problem, which could cause deadlock, we must consider the resource ordering.

7.2 Dynamic priority ordering

If nothing is locked yet, then we know that it is always safe to lock a resource. Furthermore, if it is safe to lock a resource at index j and we know that a resource at index i has lower priority, it is safe to lock i . In this way, we can construct a chain of priority ordering proofs. First, `LTFin` gives the result of an *informative* comparison of `Fins`:

```

data LTFin : Fin n → Fin n → ★ where
  ltO : LTFin f0 (fs k)
  | ltS : LTFin x y → LTFin (fs x) (fs y)

```

Using the knowledge that i is lower priority than j , we can construct the proof that it is safe to lock by induction over the instance of `LTFin`:

```

lockEarlier : LTFin i j → PriOK j xs → PriOK i xs
lockEarlier ltO locked ↪ isFirst
lockEarlier (ltS p) (isLater locked)
  ↪ isLater (lockEarlier p locked)

```

The resources could appear in any order at run-time, so we require a comparison operation which provides the appropriate ordering proof:

```

data CmpFin : Fin n → Fin n → ★ where
  lSmall : LTFin x y → CmpFin x y
  | rSmall : LTFin y x → CmpFin x y
  | finEq : (x = y) → CmpFin x y
cmpFin : (x : Fin n) → (y : Fin n) → CmpFin x y

```

The `cmpFin` function is implemented in the obvious way by recursion over x and y .

7.3 A correct implementation

To correct the possible deadlock, we must first check, dynamically, which resource has the higher priority. Using the informative comparison `cmpFin` means that we will obtain an ordering proof which can be given to `lockEarlier` to give the required priority ordering proof.

We can construct these proofs dynamically, given the resource ordering. We write a helper, `moveMoney'`, as follows, initially considering only the case where the receiver is higher priority:

```

moveMoney' : Int →
  (sender : Fin n) → (receiver : Fin n) →
  (ord : CmpFin sender receiver) →
  Lang (accounts n) (accounts n) TyUnit
moveMoney' sum sender receiver (lSmall p)
  ↪ do let sendEl = elemIs sender -
      let recvEl = elemIs receiver -
      LOCK recvEl □1
      LOCK sendEl □2
      sendFunds ← READ sendEl
      recvFunds ← READ recvEl
      CHECK (isLT sendFunds sum)
      (ACTION (putStrLn "Insufficient funds"))
      (λ p. do WRITE sendEl (sendFunds - sum)
          WRITE recvEl (recvFunds + sum)
          UNLOCK sendEl
          UNLOCK recvEl)

```

To fill in \square_2 , we use the knowledge that `receiver` is a lower priority than `sender`, and apply `lockEarlier` to the proof of \square_1 .

```

□2 = lockEarlier p □1

```

The proof of \square_1 is simple, because nothing is yet locked. We use the following a lemma to construct a proof that the first resource we lock has a valid priority:

```

unlockedAcc : (i : Fin n) → PriOK i (accounts n)
unlockedAcc f0 ↪ isFirst
unlockedAcc (fs k) ↪ isLater unlockedAcc

```

Then to fill in \square_1 we simply apply the lemma:

```

□1 = unlockedAcc receiver

```

The other case proceeds similarly. We must also consider the case where the sender and receiver are the same — there is no static check to prevent this, but in this case the function need not do anything. We can complete `moveMoney` as follows:

```

moveMoney : Int →
  (sender : Fin n) → (receiver : Fin n) →
  Lang (accounts n) (accounts n) TyUnit
moveMoney sum s r
  ↪ moveMoney' sum s r (cmpFin s r)

```

This function is guaranteed to lock accounts before use, to be deadlock free, and to release any resources it uses before exit whether an error occurs or not.

7.4 Code reuse

This implementation, in which we implement `moveMoney` twice, once for each priority ordering, highlights an inconvenient but not insurmountable problem with code reuse in currently available dependently typed programming tools. The body of the function is the same in each case, reading and writing the sum of money,

but the types and required proofs are different. We can, of course, as normal, lift out the body into a common polymorphic auxiliary function parametrised over the necessary proofs. However, this requires the programmer to pay too much attention to the intermediate types. These intermediate types (which express, for example, the information about which resources are currently locked) are inferred by unification in the implementation above. Unfortunately, since full type inference for dependently typed languages is not possible, lifting the body out into a separate function means that the *programmer* is required to work out the intermediate type, rather than the typechecker. In this kind of situation, we expect that in a more mature programming language the typechecker will be able to perform some limited type inference on auxiliary functions.

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 [21] 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 we effectively place the permissible states in the types of the DSEL. 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 are not limited to a predetermined number of states — the states we need (such as the types and values of shared variables, or the depth of locking) can be decided at run-time.

Various special-purpose type systems have also been developed. For example [32] can be used to enforce security properties and [17, 28] can be used to prevent deadlock and race conditions. Igarashi and Kobayashi have also developed a type system [15] for guaranteeing resource properties. While these approaches seem promising as specialised applications, we prefer to build on a strong general-purpose type system, thus avoiding the need to develop new soundness proofs and a new type checking implementation. Our approach, in separating the generic and specific components of a DSEL as described in section 6.7, allows *composable* domain-specific embedded languages potentially in several domains.

An obvious approach to dealing with resources is to use linear types, e.g. [14, 13]. 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 management of resources.

An alternative but related approach involves exploiting program monitoring [19] 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.

There are approaches to concurrent programming which are not based on our traditional view of locks on shared resources. For example, software transactional memory [12] works in terms of *transactions* without regard to what other threads may be doing, with a final *commit* which validates and applies transactions. Erlang³ follows the actor model [1] in which processes send messages rather than share data. It may be interesting in future to implement (and therefore verify) such approaches using our DSEL.

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) [24] in Haskell or Omega [27]. Swierstra and Altenkirch give a functional semantics to concurrency in [30] which allows reasoning about impure programs, which way may also be able to exploit with dependent types. The real benefit we obtain from using full dependent types is the ability to lift arbitrary values and functions directly into types, thus giving us the ability to refer to state directly.

9. Conclusion

Concurrent programming is becoming increasingly important, as multi-core architectures become increasingly common. As a result, it is important that programming languages provide sufficient support to allow programmers to take full advantage.

We have shown how to develop Domain-Specific Embedded Languages in a dependently typed meta-language, IDRIS, which are capable of statically guaranteeing correct resource usage (with respect to access and deadlock prevention) *by construction*. We have separated the specific state handling constructs from the generic constructs, meaning that our approach is applicable in any situation in which the effects of each operation on state can be expressed. This could cover, for example, file management operations, some aspects of network protocols, or memory consumption.

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

- No two threads will access a resource simultaneously, since a resource access must be accompanied by a proof that the process has the lock on the resource.
- Resources cannot be requested in an order which may lead to deadlock. Although it is undecidable in general whether a program will deadlock, by eliminating one of the necessary conditions we can conservatively guarantee it will never occur.
- All resources are released on exit from a process.

In developing this approach to Domain-Specific Embedded Language implementation, we have undertaken a substantial exercise in dependently typed programming. We have exploited the strengths of dependent types, in particular the strong correctness properties, but have also identified some weaknesses in current tools. In particular, code reuse can be difficult. Code which is *syntactically* the same in more than one place, such as the body of each branch of `moveMoney'`, nevertheless can have a different *type*. Lack of full type inference means that the programmer must write down such types, which can be complex. It may therefore be beneficial to identify circumstances where types can be inferred, for example following the bidirectional typechecking ideas of [20]. Furthermore, we often find ourselves implementing the same data structure several times, with different indices, e.g. `Vect` and `Env` are both lists. It would be preferable to implement functions over such structures once, generically. Nevertheless, if we do want complete static safety, we should not expect it to come totally for free.

We have tried as far as possible to push the safety guarantees into the DSEL implementation, so that the programmer can concentrate on the details of the specific problem, rather than detailed correctness proofs. However, requiring programs to be provably deadlock free means that an application programmer must think about *why* resources are requested in the right order. Where resources are statically known, the typechecker can infer the relevant proofs, but where resources are dynamic such as our bank account example, some simple reasoning is required. The DSEL author can help by providing a library of useful lemmas, such as `lockEarlier` and a generic version of `unlockedAcc`. In all cases, the application programmer is able to construct such ordering proofs dynamically

³<http://www.erlang.org/>

— the typing rules will still ensure that deadlock does not occur, since they ensure that any necessary checks are executed.

9.1 Future work

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 FORK but would then also need to deal with synchronisation and communication between threads.

Secondly, modern operating-system kernels require similar locking operations. On 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 kernel and driver programmers.

Thirdly, we believe our approach to be applicable in a number of other contexts. One area we have begun to investigate is network protocol correctness. An 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 [5], 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] G. Agha, I. Mason, S. Smith, and C. Talcott. A foundation for actor computation. *Journal of Functional Programming*, 7(1):1–72, 1997.
- [2] P. Amey. Correctness by Construction: Better can also be Cheaper. *CrossTalk: the Journal of Defense Software Engineering*, pages 24–28, Mar. 2002.
- [3] E. Brady. *Practical Implementation of a Dependently Typed Functional Programming Language*. PhD thesis, University of Durham, 2005.
- [4] E. Brady. Ivor, a proof engine. In *Implementation and Application of Functional Languages 2006*, volume 4449 of LNCS. Springer, 2007.
- [5] E. Brady and K. Hammond. A dependently typed framework for static analysis of program execution costs. In *Proc. Implementation of Functional Languages (IFL 2005)*, volume 4015 of LNCS, pages 74–90. Springer, 2006.
- [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] E. Brady, C. McBride, and J. McKinna. Inductive families need not store their indices. In *Types for Proofs and Programs 2003*, volume 3085. Springer, 2004.
- [8] E. Coffman, M. Elphick, and A. Shoshani. System deadlocks. *ACM Computing Surveys*, 3(2):67–78, 1971.
- [9] K. Czarnecki, J. O'Donnell, J. Striegnitz, and W. Taha. DSL implementation in MetaOCaml, Template Haskell, and C++. In *Domain Specific Program Generation 2004*, volume 3016 of LNCS. Springer, 2004.
- [10] K. Hammond and G. Michaelson. Hume: a Domain-Specific Language for Real-Time Embedded Systems. In *Proc. Conf. Generative Programming and Component Engineering (GPCE '03)*, Lecture Notes in Computer Science. Springer-Verlag, 2003.
- [11] 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.
- [12] T. Harris, S. Marlow, S. Peyton Jones, and M. Herlihy. Composable memory transactions. In *PPoPP'05: ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2005.
- [13] C. Hawblitzel. Linear types for aliased resources. Technical Report MSR-TR-2005-141, Microsoft Research, 2005.
- [14] M. Hofmann and S. Jost. Static prediction of heap space usage for first-order functional programs. In *Proc. POPL 2003 — 2003 ACM Symp. on Principles of Programming Languages*. ACM, 2003.
- [15] A. Igarashi and N. Kobayashi. Resource usage analysis. In *Symposium on Principles of Programming Languages*, pages 331–342, 2002.
- [16] 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*, Washington, DC, USA, 2005.
- [17] N. Kobayashi. A type system for lock-free processes. *Inf. Comput.*, 177(2):122–159, 2002.
- [18] P. Landin. The next 700 programming languages. *Communications of the ACM*, 9(3), March 1966.
- [19] J. A. Ligatti. *Policy Enforcement via Program Monitoring*. PhD thesis, May 2006.
- [20] A. Löb, C. McBride, and W. Swierstra. Simply easy! — an implementation of a dependently typed lambda calculus, 2007. Draft.
- [21] 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.
- [22] C. McBride and J. McKinna. The view from the left. *Journal of Functional Programming*, 14(1):69–111, 2004.
- [23] E. Pašalić, W. Taha, and T. Sheard. Tagless staged interpreters for typed languages. In *Proc. 2002 International Conf. on Functional Programming (ICFP 2002)*. ACM, 2002.
- [24] 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.
- [25] 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.
- [26] C. Popea 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.
- [27] T. Sheard. Languages of the future. In *ACM Conference on Object Oriented Programming Systems, Languages and Applications (OOPSLA'04)*, 2004.
- [28] K. Suenaga and N. Kobayashi. Type-based analysis of deadlock for a concurrent calculus with interrupts. 2007.
- [29] H. Sutter. Use lock hierarchies to avoid deadlock. *Dr Dobb's*, December 2007.
- [30] W. Swierstra and T. Altenkirch. Beauty in the beast: A functional semantics of the awkward squad. In *Haskell '07: Proceedings of the ACM SIGPLAN workshop on Haskell*, 2007.
- [31] W. Taha. *Multi-stage Programming: Its Theory and Applications*. PhD thesis, Oregon Graduate Inst. of Science and Technology, 1999.
- [32] D. Walker. A type system for expressive security policies. In *Twenty-Seventh ACM SIGPLAN Symposium on Principles of Programming Languages*, pages 254–267, 2000.