Fundamenta Informaticae XX (2009) 1–31
IOS Press

Correct-by-Construction Concurrency: using Dependent Types to Verify Implementations of Effectful Resource Usage Protocols

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, including locks and shared variables, can be especially difficult and error-prone. A simple mistake, such as forgetting to release a lock, can have major consequences on the correct operation of a program, by, for example, inducing deadlock, often at a time and location that is isolated from the original error. 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 directly enforce the formal program properties that we require. In this way, we ensure strong static guarantees of *correctness-by-construction*, without requiring the development of a new special-purpose type system or the associated special-purpose soundness proofs. We also reduce the need for "over-serialisation", the overly-conservative use of locks that often occurs in manually constructed software, where formal guarantees cannot be exploited. We illustrate our approach by implementing a DSEL for concurrent programming and demonstrate its applicability with reference to an example based on simple bank account transactions.

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 [23], where any thread may initiate an I/O request in any required order, is essential to provide high performance in this new setting. However, locking and other concurrency issues make this a difficult and error-prone environment for the typical programmer. In order to guarantee safety, it is common practice to use locks wherever the programmer is uncertain about concurrent access to a resource (even where they *are* certain that there is no concurrent access, programmers may still include locks in order to ensure robustness in the face of future software changes). 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 programming abstraction will both ensure safety *and* minimise the frequency of locking and other overhead costs. In this way, we aim to achieve "correctness-by-construction" [2] for concurrent systems, where any valid concurrent program is formally guaranteed to possess certain required properties. Using such an approach both improves software reliability and reduces development costs, through eliminating unnecessary testing and debugging steps. This paper therefore studies how "true concurrency" can be achieved in a sound way, using verified, correct-by-construction software implementing the required lock access policy.

Popeea and Chin have previously observed [42] that there are, in fact, 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, 7]. 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, and so construct a simple dependently-typed Domain-Specific Embedded Language (DSEL) that captures locking information. The typing rules of this DSEL then directly enforce the required safety properties, including the safe acquisition and release of resource locks.

1.1. Contributions of this Paper

In illustrating a type-based approach to resource usage verification, this paper makes a number of novel contributions. It directly addresses language-based safety and security issues in a multi-threaded environment with shared state, considering relevant theoretical and practical considerations for controlling resource locks on multi-threaded systems. The three main technical contributions made here are:

1. We give a new framework for state-handling domain-specific embedded languages (DSELs) using a dependently typed host 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: i) that all resources are requested before use; ii) that all acquired resources are released before program exit; and iii) that deadlock does not occur (Sections 4–6).

As discussed above, there is a danger of *over-serialising* concurrent software, where a programmer lacks confidence that basic safety properties are satisfied and therefore uses locks in an overly cautious way. A consequence of our type-based approach is that the type system gives the programmer confidence that these properties hold and thus helps prevent such over-serialisation.

In addition to these technical contributions, this paper introduces the IDRIS dependently typed programming language (`http://www.cs.st-and.ac.uk/~eb/Idris/`). IDRIS is intended as a platform for experimentation with practical aspects of dependently typed functional programming, such as the

interaction of dependent types with I/O, shared external state and concurrency used here. 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

A major reason that concurrent software is often difficult both to implement correctly, and to prove correct, is because the order in which processes are executed can be hard or even impossible to determine. There are potential difficulties with even apparently simple code. For example, there are (at least) two problems with the following very simple concurrent pseudo-code procedure, which aims to move a sum of money from one bank account to another. This procedure requests a lock on the resource associated with each account, to prevent other threads accessing the account in the critical section between the read and the write. It then moves the money between the accounts and unlocks the resources. Only one thread may hold a lock at any time. If another thread holds the lock, this thread will block until it is released.

```
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);
}
```

The first problem occurs if there is insufficient funds in the sender's account. In this case, the `moveMoney` function reports an error and then returns, but fails to unlock the resources it has acquired. This has disastrous consequences, since now no other process can ever access either account! This is a common, and well-understood problem, but one that still occurs surprisingly often, and which can be very hard to correct (for example, the original Java approach to *resource finalisers* has been deprecated precisely because it is difficult to implement safely in the presence of threads!). The second problem is a little harder to spot. Imagine that the following calls are made simultaneously in three separate processes:

1. `moveMoney(20, Thorsten, Tarmo)`
2. `moveMoney(10, Tarmo, Per)`
3. `moveMoney(42, Per, Thorsten)`

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

1. Thorsten sends Tarmo £20, beginning by executing `lock(Thorsten)`.
2. Tarmo sends Per £10, beginning by executing `lock(Tarmo)`.

3. Per sends Thorsten £42, beginning by executing `lock(Per)`.

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 is *caused* by the code — no attempt has been made to prevent deadlocks such as this occurring[1].

Concurrent systems such as the one above are naturally and inherently stateful, and their correct operation relies on it being impossible for them to enter invalid states, such as attempting to return from a process without releasing resources or requesting system resources in the wrong order. In traditional imperative programming languages, such properties cannot in general be checked mechanically — for most systems, correctness relies either on informal coding conventions, the deployment of a costly run-time monitor, or explicit dynamic checking; or in those cases where safety is paramount, perhaps the use of *model checking* or some other formal method. 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, purely functional programming can provide a means for checking properties such as those described above *statically* and *within the implementation itself*, thereby guaranteeing that resources are requested and released as necessary without the risk of causing deadlock. We will achieve this by implementing a domain specific embedded language (DSEL) within a host dependently typed language.

3. Dependently Typed Programming

IDRIS is a full-spectrum dependently typed programming language, similar to EPIGRAM [33, 34] or AGDA[2], built on top of the IVOR [5] theorem proving library. It is a purely functional language with a syntax similar to Haskell plus General Algebraic Data Types (GADTs) [40]. By default, all IDRIS functions are *total*: this is guaranteed in the manner suggested by Turner [47], by requiring that all possible cases are covered by every definition, and that recursive calls proceed on structurally smaller values of *strictly positive data types*. This restriction is required to preserve decidability of type-checking, and to ensure *total* correctness of proofs, but can be relaxed by the programmer if *partial* correctness only is required. The purpose of the IDRIS 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 us complete freedom to experiment with abstractions and language features beyond the type system, such as I/O and concurrency. In addition, although unrelated to the work that we present in this paper, the core language of IDRIS is intended as an important step towards a fully dependently typed implementation of the real-time Hume language [15]. In this section, we introduce some features of IDRIS and describe some important basic techniques in dependently typed programming. A more detailed introduction to dependently typed programming can be found elsewhere [33].

[1] A possible, widely-used, solution to this problem is to impose a priority ordering on resources (for example, account number) and to always lock the higher priority resource first. We will return to this later.
[2] `http://wiki.portal.chalmers.se/agda/`

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 providing *informative* results of tests. 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
(!!) (x:xs) 0 =  x
(!!) (x:xs) n =  (!!) xs (n-1)
```

(where `xs!!0` returns the first element of list `xs`, and so on). Suppose we use this function as follows:

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

Since we have made a choice based on the ith 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; and, as programmers, we do, of course, regularly make such assumptions based on our own knowledge of the code. The compiler, however, has no such knowledge and therefore cannot guarantee that the assumptions that we have made are valid. We consequently find out about our mistakes only when we encounter a run-time error. In general-purpose systems, this is merely inconvenient and painful. In safety-critical systems, this may be literally fatal. *Dependent types allow us to avoid such problems by permitting us to explicitly state, and implicitly enforce, our assumptions.*

For example, consider the following data type representing vectors (sized lists):

$$\begin{array}{l}\underline{\mathsf{data}}\quad \mathsf{Vect}\ :\ \star \to \mathbb{N} \to \star\quad \underline{\mathsf{where}}\\ \quad \mathsf{nil}\ :\ \mathsf{Vect}\ A\ 0\\ \mid\ \mathsf{cons}\ :\ A \to \mathsf{Vect}\ A\ k \to \mathsf{Vect}\ A\ (\mathsf{s}\ k)\end{array}$$

This declaration is written in a GADT style: we give the *type constructor* Vect, which is parameterised over a type and a natural number ($\star$ is the type of types), followed by a list of *data constructors* and their types. Corresponding to this, we have the finite sets which we can use to represent bounded numbers:

$$\begin{array}{l}\underline{\mathsf{data}}\quad \mathsf{Fin}\ :\ \mathbb{N} \to \star\quad \underline{\mathsf{where}}\\ \quad \mathsf{f0}\ :\ \mathsf{Fin}\ (\mathsf{s}\ k)\\ \mid\ \mathsf{fs}\ :\ \mathsf{Fin}\ k \to \mathsf{Fin}\ (\mathsf{s}\ k)\end{array}$$

Finite sets are parameterised over the number of values the set can contain. We can read $\mathsf{Fin}\ n$ as the type of "a number less than n." We now write the bounds-safe lookup corresponding to `!!`:

$$\begin{array}{l}\mathbf{vlookup}\ :\ \mathsf{Fin}\ k \to \mathsf{Vect}\ A\ k \to A\\ \quad \mathbf{vlookup}\ \ \mathsf{f0}\ \ (\mathsf{cons}\ x\ xs)\ \mapsto\ x\\ \quad \mathbf{vlookup}\ (\mathsf{fs}\ i)\ (\mathsf{cons}\ x\ xs)\ \mapsto\ \mathbf{vlookup}\ i\ xs\end{array}$$

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

$$\begin{array}{l}\underline{\mathsf{data}}\quad \mathsf{ElemIs}\ :\ \mathsf{Fin}\ n \to A \to \mathsf{Vect}\ A\ n \to \star\quad \underline{\mathsf{where}}\\ \quad \mathsf{first}\ :\ \mathsf{ElemIs}\ \mathsf{f0}\ x\ (\mathsf{cons}\ x\ xs)\\ \mid\ \mathsf{later}\ :\ \mathsf{ElemIs}\ i\ x\ ys \to \mathsf{ElemIs}\ (\mathsf{fs}\ i)\ x\ (\mathsf{cons}\ y\ ys)\end{array}$$

So rather than simply looking up a value, given a list and an index, we instead compute a predicate which is a proof that the element that we looked up is stored at the given index, using the following function:

$$\begin{array}{l}\mathbf{elemIs}\ :\ (i\ :\ \mathsf{Fin}\ n) \to (xs\ :\ \mathsf{Vect}\ A\ n) \to \mathsf{ElemIs}\ i\ (\mathbf{vlookup}\ i\ xs)\ xs\\ \quad \mathbf{elemIs}\quad \mathsf{f0}\quad (\mathsf{cons}\ x\ xs)\ \mapsto\ \mathsf{first}\\ \quad \mathbf{elemIs}\ (\mathsf{fs}\ k)\ (\mathsf{cons}\ x\ xs)\ \mapsto\ \mathsf{later}\ (\mathbf{elemIs}\ k\ xs)\end{array}$$

Note that in the type of **elemIs**, we name the arguments i and xs, since the return type *depends* on the value of these arguments. Names can optionally be given in other cases, even if the return type does not depend on an argument. Here, we name arguments where it helps to document an argument's purpose. IDRIS checks exhaustiveness of the patterns for **elemIs** by unification — all other possible patterns (e.g. for nil) generate a unification error because the index i (of type Fin n) and vector xs (of type Vect A n) must have the same bounds.

Depending on our application, we can use as much or as little of the information given by **elemIs** as we need. Often, simply looking up the element with **vlookup** is sufficient. However, depending on the nature of the safety requirements, 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 over the entire lifetime of the program.

3.2. Collapsible data structures

ElemIs is an example of a **collapsible** data structure [8]. 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 if 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 it follows that the indices uniquely determine the constructor. Furthermore, 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 that proofs need not be given explicitly in the type.

- At run-time, the value can be *discarded*, as detailed in [4, 8], 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 overhead of actually storing the proofs at run-time.

It is important to realise, therefore, that the presence of such proofs in our programs has absolutely *no* performance penalty.

3.3. I/O with dependent types

We require input/output operations to be executable from within our language. To achieve this, we exploit Hancock and Setzer's I/O model [16]. This involves implementing a Haskell-style IO monad [41] by defining *commands*, representing externally executed I/O operations, and *responses*, which give the type of the value returned by a given command. Compiling I/O programs via C then simply involves translating the commands into their C equivalents. IDRIS implements a simplified form of do notations which translates directly to **bind** and **return** functions in the IO monad.

For the programs we will discuss in this paper, we will require the following operations:

$$
\begin{array}{lcl}
\mathbf{fork} & : & \mathsf{IO}\ () \rightarrow \mathsf{IO}\ () \\
\mathbf{newLock} & : & \mathsf{Int} \rightarrow \mathsf{IO}\ \mathsf{Lock} \\
\mathbf{lock}, \mathbf{unlock} & : & \mathsf{Lock} \rightarrow \mathsf{IO}\ () \\
\mathbf{newIORef} & : & A \rightarrow \mathsf{IO}\ (\mathsf{IORef}\ A) \\
\mathbf{readIORef} & : & \mathsf{IORef}\ A \rightarrow \mathsf{IO}\ A \\
\mathbf{writeIORef} & : & \mathsf{IORef}\ A \rightarrow A \rightarrow \mathsf{IO}\ ()
\end{array}
$$

Lock is an externally implemented semaphore, created by **newLock**, with **lock** and **unlock** requesting and releasing the semaphore. The other functions behave in exactly the same way as their Haskell equivalents: forking new processes; and creating, reading from and writing to reference locations. Our implementation is built on the pthreads library, an implementation of the POSIX threads standard [9].

4. Resources for Concurrent Programming

We have seen above that dependent types offer a way to specify and to verify important program properties. However, in practice, many realistic problems require interaction with the outside world and manipulation of state. Our goal is to capture this state management in a program's type, and use the power of dependent types to verify the correct management of low-level resources in concurrent software, without unnecessarily restricting any other operations which have no impact on resource management. In this section, we consider the resources we wish to manage, the properties which those resources should maintain, and how to formalise those properties.

4.1. Ensuring Safe Resource Access

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

$$
\begin{array}{lcl}
\mathbf{increment} & = & \underline{\mathsf{do}}\ val \leftarrow \mathsf{READ}\ var \\
& & \quad \mathsf{WRITE}\ (val + 1)\ var
\end{array}
$$

Clearly, the correct outcome is for the 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 rather than by two! 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 from occurring by ensuring that all shared variables are locked before they are accessed.

4.2. Preventing Deadlock

A second challenge that occurs in more complex concurrent programs is deadlock avoidance: namely preventing a situation where two or more threads are waiting for each other to release some resource. Determining whether deadlock occurs is, in general, undecidable. If we want a sound and complete approach to proving the absence of deadlock statically, we could define a type which specifies deadlock freedom and would need to prove by hand that a concurrent program has that type by reasoning about all possible execution orders. However, if we want to avoid such complexity in the proofs of deadlock freedom, it will be necessary to restrict the system in some way.

The approach we have chosen still requires some hand written proofs, but avoids the complexity of reasoning about all possible execution orders, at the expense of being incomplete (i.e. rejecting some deadlock-free programs). It is based on the observation that there are four necessary conditions for a deadlock to occur [10]:

1. Resources are mutually exclusive, i.e. only one process at a time can access the resource;
2. Processes may request the use of additional resources;
3. Only a process with access to a resource may release it; and
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 four conditions from occurring, then deadlock can never occur. Clearly, the first three will apply in any concurrent system which requires access to shared resources. We aim to prevent deadlock by enforcing an *ordering* on resources, and so eliminating condition 4. Although there is not always an obvious ordering that can be exploited, this method has proven to be useful in practice [45]. We therefore prevent circular chains from forming by requiring that each resource that a process requests has a *lower* priority than any of those that it already holds.

Such difficulties are, of course, dealt with on an everyday basis by concurrency practitioners, who obtain working solutions for their programs by employing a variety of informal coding conventions. However, such conventions are difficult to enforce in a safe and secure way across the entire community. Moreover, the use of locking and unlocking operations is potentially highly error-prone, particularly in situations where threads may 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 the one we have described above can *never* be encountered.

4.3. Formalising Resource Management

We have two main options for formalising the above properties:

1. Define and prove sound a special purpose type system which captures these properties, and the pre- and post-conditions on the required operations. From such a type system, we can extract an implementation which statically checks the correct usage of resources.

2. Exploiting dependent types, implement the operations in IDRIS, and write the pre- and post-conditions in the *type* of each operation.

We have preferred the second approach, for two main reasons. Firstly, we contend that dependent type theory is strong enough to specify the required properties formally and precisely. We argue that the soundness of the underlying type system guarantees the correctness of the operations. Secondly, we are not limited to the operations we choose to define in a special purpose type system, and can use the full power of the implementation language. Additionally, we would like to explore the strengths and limitations of dependent types for extra-functional correctness.

Like any formally defined system, the correctness relies on capturing the necessary properties accurately. An advantage of using dependent type theory to capture the properties is that the implementation and the formalisation itself are the *same thing*. A dependently typed language gives us exactly the notation we need to construct a machine-checkable formalisation of the operations we implement.

4.4. The Domain-Specific Embedded Language Approach

A domain-specific language (DSL) is a programming language dedicated to a specific problem domain. Using a DSL, a **domain expert**, who is not necessarily a computer programmer, can focus on details of the problem to be solved, rather than on details of the implementation language. A domain-specific *embedded* language [20] (DSEL) is a DSL that has been implemented by embedding the DSL constructs into another language, the *host* language. It is important to maintain a clear distinction between the implementation language (the host language, here IDRIS) and the language being implemented (the DSEL). The DSEL approach allows the language implementer to avoid dealing with well understood aspects of language implementation such as parsing and code generation, and a certain amount of type checking [20]. It is a commonly applied technique in the Haskell community. Well-known examples include Parsec [28] and Yampa [12]. Dependent types allow us to take this idea further, using the host language to express strong guarantees about the DSEL.

5. State-Handling Domain-Specific Embedded Languages

In this section we consider how to implement a state-handling DSEL in IDRIS. We will introduce the language for concurrency management in two stages:

- Firstly (in this section), we will show the general form of a state-handling DSEL.

- Secondly (in Section 6), we will specialise the language to the concurrency domain, by considering the state of shared resources.

In general, the programs we consider will manipulate state (i.e., have side effects) and compute a result. Since our goal is static safety, we will, as far as possible, 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.

5.1. Types

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

```
data Ty = TyUnit | TyLift ⋆
```

DSEL types can be converted to host language types in the obvious way:

```
interpTy : Ty → ⋆
interpTy TyUnit      ↦ ()
interpTy (TyLift A)  ↦ A
```

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 host language types) because it fulfills a special rôle in the DSEL, and also because it maintains a clear logical separation between host language types (of type $\star$) and DSEL types (of type Ty).

5.2. Language Representation

A DSEL program modifies state, and returns a value. Our representation should therefore reflect this:

```
data Lang : StateIn → StateOut → Ty → ⋆
```

We leave the definitions of $StateIn$ and $StateOut$ open for the moment, since these will depend on the specific domain. For complete flexibility, we will 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 involves modifying its type.). Figure 1 gives declarations for the common DSEL constructs.

```
data Lang : StateIn → StateOut → Ty → ⋆ where
    LOOP   : (count : Nat) → (body : Lang s_in s_in TyUnit) → Lang s_in s_in TyUnit
  | CHECK  : Maybe a → (ifJ : a → Lang s_in s_out ty) → (ifN : Lang s_in s_out ty) →
             Lang s_in s_out ty
  | ACTION : IO () → Lang s_in s_in TyUnit
  | RETURN : interpTy ty → Lang s_in s_in ty
  | BIND   : (code : Lang s_in s_k ty) → (k : interpTy ty → Lang s_k s_out tyout) →
             Lang s_in s_out tyout
```

Figure 1. Common DSEL constructs

Each constructor of the Lang data type corresponds to a syntactic construct in the DSEL. Since the representation of the syntax is parameterised over types, this also means that we can express the DSEL's typing rules in the constructors. For example, when given a value in the host language, RETURN constructs a value in the DSEL. The corresponding constructor in the Lang type reflects this:

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

A second, equally important, feature of the host language 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:

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

Even though our goal is total static safety, dynamic checking of values may still be required in some cases, and in particular we may need to construct proofs dynamically. We therefore include a CHECK construct, which takes the result of a host language computation, of type Maybe a, and continues execution according to the result. Maybe a is an option type, which either carries a value x of type a (Just x), or no value (Nothing). Typically, the a will 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 any necessary dynamic checks.

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

Embedding a domain-specific language within a host language allows us to exploit the features of the host language. For example, we can allow IO actions to be executed as follows:

$$\mathsf{ACTION} \ : \ \mathsf{IO} \ () \rightarrow \mathsf{Lang} \ s_{in} \ s_{in} \ \mathsf{TyUnit}$$

We have also included a bounded looping construct, which allows an expression to be evaluated $count$ times. In principle, we could include any control structure which is implementable in IDRIS. LOOP requires that the state is the same on entry and exit of each iteration:

$$\mathsf{LOOP} \ : \ (count \ : \ \mathsf{Nat}) \rightarrow (body \ : \ \mathsf{Lang} \ s_{in} \ s_{in} \ \mathsf{TyUnit}) \rightarrow \mathsf{Lang} \ s_{in} \ s_{in} \ \mathsf{TyUnit}$$

Remark: This LOOP construct implies that a DSEL implemented in this way cannot handle non-terminating programs. Since IDRIS requires that functions be total, in order to guarantee total correctness and decidability of type checking as discussed in Section 3, it is not possible to implement an unbounded looping construct (e.g. a while loop) directly. Although we have not done so, we can choose to implement an unbounded loop by separating *evaluation* from *execution*, as discussed in [16]. The details are beyond the scope of this paper.

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

6. A DSEL For Concurrent Programming

In this section we will complete our DSEL for concurrent programming by defining: i) the state over which the DSEL is parameterised; and ii) those operations that affect this state. We define operations for forking new processes, managing locks on shared variables, and reading and writing shared variables. For each of these operations we define the pre- and post-conditions *informally*, as described in Section 4, and show how to encode those properties *formally* in the host language.

6.1. Resources

We begin by defining formally the state over which the DSEL is parameterised, and the properties the state can satisfy. The resources which interest us are shared typed variables, which must be locked before they are accessed (read or written). In formalising lock management, we therefore need to define the lock state of a resource, and the type of the value associated with the resource. We choose to allow locks to be *nested*, so the lock state also needs to represent the number of times each resource has been locked:

```
data ResState : ⋆ where
  RState : ℕ → Ty → ResState
```

Resource state is *per thread*. When a resource is locked, thread execution can only proceed if no other thread currently holds that lock. Otherwise, execution blocks until the lock is released. Nesting locks means that one thread may lock a resource multiple times, and the resource will only become free when the lock count reaches zero. This behaviour models what happens at the operating system level, using pthreads.

Furthermore, when we run the program, we will need access to the *concrete data* that is associated with the resource. In our case, this concrete data is the semaphore that controls access to the resource (of type Lock, as defined in Section 3.3), plus the value itself. Since a value is a piece of mutable external state, we must store it in an IORef. The Resource type is parameterised 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.

```
data Resource : ResState → ⋆ where
  resource : IORef (interpTy a) → Lock → Resource (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: resources can then be easily referred to by defining an index into this Vect. The index is stored as a Fin n where n is the number of resources. We can now define $StateIn$ and $StateOut$ in concrete terms:

```
data Lang : Vect ResState tin → Vect ResState tout → Ty → ⋆
```

This type declaration gives us the potential to create or destroy resources within a DSEL program. In this paper, however, we assume that the resources are known in advance. It turns out that this restriction is in fact minor: while resources cannot be created *within* the DSEL, they can be freely constructed dynamically on an *ad-hoc* basis in the host language, and subsequently *managed* in a thread-safe manner by passing them to a DSEL program. This follows common practice [38, 45], where choosing an ordering for requesting locks and using it consistently prevents circular chains of requests. The dependent type system allows us to guarantee this ordering is preserved.

6.2. Resource Validity

To prevent deadlock, as we have discussed in Section 4.2, we can define a priority ordering on resources, and prove that higher priority resources are always locked earlier than lower priority resources. Storing resources as a vector gives a convenient ordering, namely the position of the resource in the vector. We treat resources which appear earlier in the vector as lower priority. In order for the static property to hold, we must then prove that whenever we lock a resource, no lower priority resource has already been locked. The predicate PriOK enforces this property. It can be constructed if, and only if, everything before a given index in a vector is an unlocked resource.

```
data PriOK : (i : Fin n) → (xs : Vect ResState n) → ⋆ where
     isFirst : PriOK f0 (cons x xs)
   | isLater : PriOK i xs → PriOK (fs i) (cons (RState 0 t) xs)
```

It is worth considering briefly how we arrive at this definition. Consider first how we might check PriOK dynamically, returning a truth value. It is always safe to lock the first resource, since it has the lowest possible priority. Clearly, any non-first resource can only be locked if the first resource (i.e., the one with the lowest priority) is unlocked:

```
isPriOK : Fin n → Vect ResState n → Bool
isPriOK f0       (cons x xs)          ↦ True
isPriOK (fs i) (cons (RState 0 t) xs) ↦ isPriOK i xs
isPriOK _        _                    ↦ 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. It follows that arriving at the definition of such a predicate corresponds directly to implementing the relevant dynamic check. In fact, we can make this test more informative as follows:

```
isPriOK : (i : Fin n) → (xs : Vect ResState n) → Maybe (PriOK i xs)
isPriOK f0       (cons x xs)          ↦ Just isFirst
isPriOK (fs i) (cons (RState 0 t) xs) ↦ mMap isLater (isPriOK i xs)
isPriOK _        _                    ↦ Nothing
```

Where we previously returned False, we now return Nothing. In addition, instead of simply returning True to indicate that the priority is okay, we now also *explain why it is okay*. 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. Forking a process

When we fork a new process, the resource state of the new process holds no locks. We express this statically, with a predicate:

```
data  AllUnlocked : (xs : Vect ResState n) → ⋆  where
    nilUn  : AllUnlocked nil
  | consUn : AllUnlocked xs → AllUnlocked (cons (RState 0 t) xs)
```

In most cases, proofs of this predicate will be of the form consUn (consUn ... nilUn), corresponding to the number of locks, with the type checker being able to fill in extra details by unification. We have a partial decision procedure for this predicate:

```
allUnlocked : (xs : Vect ResState n) → Maybe (AllUnlocked xs)
```

If xs is statically known, and all elements are unlocked, this function computes a proof of AllUnlocked xs. While this is a common case, other more complex cases will require hand written proofs. The restriction is necessary to preserve safety: once a child process has been forked, it has no knowledge of lock state transitions in its parent. If we allow a forked process to use a resource locked by its parent, but the parent releases the lock before the child uses it, then we have violated a safety condition. Furthermore, allowing this would mean a resource was locked simultaneously by multiple threads! We add the following constructor to Lang to capture this restriction:

```
FORK : AllUnlocked tins → (proc : Lang tins tins TyUnit) → Lang tins' tins' TyUnit
```

6.4. The Full DSEL

We are now in a position to add the constructors for the domain-specific operations, which are FORK, LOCK, UNLOCK, READ and WRITE. For each of these, we will given informal pre- and post-conditions before giving a formal definition in IDRIS. Our approach is related to Hoare Type Theory [36] in that we express these conditions on each operation, but has the important distinction that we impose domain-specific conditions on domain-specific operations, which allows the DSEL programmer to concentrate on the domain-specific correctness properties.

To lock a resource, we must know that it is safe to lock in the current state. This requires the conditions:

Pre-condition The resource we lock is lower priority than any currently locked resources.

Post-condition For a resource with k nested locks, after locking, it has $k + 1$ nested locks.

We add the following constructor to Lang which expresses these conditions formally:

```
LOCK : ElemIs i (RState k ty) tins → (PriOK i tins) →
       Lang tins (update i (RState (s k) ty) tins) 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: it is convenient to be able to lift this into the type of LOCK:

```
update : (i : Fin n) → A → Vect A n → Vect A n
```

We can arrive at the necessary proofs 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. It has the following conditions:

Pre-condition The resource to be unlocked has $k+1$ nested locks (i.e. it has been locked by the current thread).

Post-condition For a resource with $k+1$ nested locks, after unlocking, it has k nested locks.

Stated formally as part of our DSEL, these conditions give:

```
UNLOCK : ElemIs i (RState (s k) ty) tins →
         Lang tins (update i (RState k ty) tins) TyUnit
```

Once again, the UNLOCK operation updates the state, this time by reducing the lock count. The purpose of locking a resource is, of course, to prevent incorrect accesses to and modifications of 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. The conditions are:

Pre-condition The resource to be read from (or written to) has $k+1$ nested locks (i.e. it has been locked by the current thread).

Post-condition None.

Stated formally as part of our DSEL, these conditions give:

```
READ  : ElemIs i (RState (s k) ty) tins → Lang tins tins ty
WRITE : interpTy ty → (ElemIs i (RState (s k) ty) tins) → Lang tins tins TyUnit
```

Neither of these operations modifies the resource state — the state carries only how many 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. Such proofs can be constructed dynamically or statically, as required.

Figure 2 gives the complete language representation. By using the DSEL approach, and considering each operation in turn we are able to give our intuition for the correctness requirements of each rule, before formalising our intuition in the host language itself. We believe that a significant advantage of the dependently typed approach is that the implementation and formalisation are the *same thing*, and that we can consequently read typing rules of the DSEL directly from the Lang data type.

6.5. Executing the DSEL

Of course, while theoretically interesting, our DSEL would be of no practical use unless we had a means of actually executing programs. Fortunately, implementing the interpreter is largely straightforward and consists of executing the relevant low-level operations for each command.

Environments

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

```
data Env : (R : ⋆) → (iR : R → ⋆) → (xs : Vect R n) → ⋆ where
     Empty : Env R iR nil
   | Extend : (res : iR r) → Env R iR xs → Env R iR (cons r xs)
```

```
data Lang : Vect ResState tin → Vect ResState tout → Ty → ⋆ where
     READ   : (locked : ElemIs i (RState (s k) ty) tins) → Lang tins tins ty
  |  WRITE  : 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   : AllUnlocked tins → (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 2. Concurrency DSEL syntax representation

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

```
envLookup : (i : Fin n) → Env R iR xs → iR (vlookup i xs)
envLookup f0   (Extend t env) ↦ t
envLookup (fs i) (Extend t env) ↦ envLookup i env
```

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

```
REnv : (xs : Vect ResState n) → ⋆
REnv xs ↦ Env ResState Resource xs
```

At various points, we will need to extract references and resource locks from the environment. To lookup a resource lock, using **llookup**, we simply take an index into the environment. To obtain the reference, using **rlookup**, we also take the ElemIs proof that a value exists at the index. Although not strictly necessary, type-checking applications of **rlookup** is assisted if the type of the resource is named.

```
llookup : {xs : Vect ResState ln} → (i : Fin ln) → REnv xs → Lock
rlookup : {xs : Vect ResState ln} → (p : ElemIs i (RState k ty) xs) → REnv xs →
          IORef (interpTy ty)
```

In the definitions of $\mathbf{llookup}$ and $\mathbf{rlookup}$, xs is an *implicit* argument. Its type is given to assist the type checker, but a value is not given explicitly in calls to these functions.

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). We assume that the concrete resources contained in the environment are *distinct*, i.e., no IORef or Lock appears more than once.

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

The type expresses that the interpreter must modify the environment to correspond exactly with any modifications 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 generally straightforward — the complete definition is given in Figure 3. The rules for LOCK and UNLOCK require the use of an implicit argument i, which is the index into the vector of resources used by the proof argument[3]. Since $\mathbf{lock}$ and $\mathbf{unlock}$ are low level operations, we also require functions $\mathbf{lockEnv}$ and $\mathbf{unlockEnv}$ to update the *type* of the environment, to make it consistent with the resource state given in the DSEL.

$$\begin{aligned}\mathbf{lockEnv} \ : \ & \{i \ : \ \mathsf{Fin}\ n\} \rightarrow \{xs \ : \ \mathsf{Vect}\ \mathsf{ResState}\ n\} \rightarrow (\mathsf{ElemIs}\ i\ (\mathsf{RState}\ k\ ty)\ xs) \rightarrow \\ & (\mathbf{REnv}\ xs) \rightarrow (\mathbf{REnv}\ (\mathbf{update}\ i\ (\mathsf{RState}\ (\mathsf{s}\ k)\ ty)\ xs)) \\ \mathbf{unlockEnv} \ : \ & \{i \ : \ \mathsf{Fin}\ n\} \rightarrow \{xs \ : \ \mathsf{Vect}\ \mathsf{ResState}\ n\} \rightarrow (\mathsf{ElemIs}\ i\ (\mathsf{RState}\ (\mathsf{s}\ k)\ ty)\ xs) \rightarrow \\ & (\mathbf{REnv}\ xs) \rightarrow (\mathbf{REnv}\ (\mathbf{update}\ i\ (\mathsf{RState}\ k\ ty)\ xs))\end{aligned}$$

Since this DSEL is implemented in a less restrictive host language, there is a chance that programs will access resource and concurrency primitives directly, rather than through the interpreter. We must therefore ensure that *only* the interpreter is allowed to access the concurrency primitives. While we have not yet implemented a module system for IDRIS, conventional techniques will allow us to hide the $\mathbf{lock}$, $\mathbf{unlock}$ and $\mathbf{fork}$ operations behind an interface which exposes only Lang and $\mathbf{interp}$.

6.6. 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:

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

It is helpful (and indeed trivial) to define a proof that all of these are initially unlocked:

$$\begin{aligned}&\mathbf{unl\_nats} \ : \ \mathsf{AllUnlocked}\ \mathbf{nats} \\ &\mathbf{unl\_nats} \ = \ \mathsf{consUn}\ (\mathsf{consUn}\ \mathsf{nilUn})\end{aligned}$$

We can then write a program (Figure 4) which loops, incrementing each variable and printing their sum on each iteration. For readability, we have used a Haskell-like do-notation. Although this is not yet implemented in IDRIS, it is a trivial transformation to insert calls to BIND in place of the do-notation.

[3] In fact, optimisation of the Lang data type [8], as implemented in IDRIS, means that i is the only argument stored for each of LOCK and UNLOCK.

```
interp : REnv ty_in → Lang ty_in ty_out T → IO (Pair (REnv ty_out) (interpTy T))
interp env (READ p)          ↦ do val ← readIORef (rlookup p env)
                                  return (MkPair env val)
interp env (WRITE v p)       ↦ do writeIORef (rlookup p env) v
                                  return (MkPair env ())
interp env (LOCK_i p pri)    ↦ do lock (llookup i env)
                                  return (MkPair (lockEnv p env) ())
interp env (UNLOCK_i p)      ↦ do unlock (llookup i env)
                                  return (MkPair (unlockEnv p env) ())
interp env (ACTION io)       ↦ do io
                                  return (MkPair env ())
interp env (RETURN val)      ↦ return (MkPair env val)
interp env (CHECK (Just a) j n) ↦ interp env (j a)
interp env (CHECK Nothing j n)  ↦ interp env n
interp env (FORK u proc)     ↦ do fork (do f ← interp env proc
                                           return ())
                                  return (MkPair env ())
interp env (BIND code k)     ↦ do coderes ← interp env code
                                  interpBind coderes k

interpBind : Pair (REnv ty_in) A → (A → Lang ty_in ty_out B) →
             IO (Pair (REnv ty_out) (interpTy B))
interpBind (MkPair env val) k ↦ interp env (k val)
```

Figure 3. The DSEL interpreter

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 do, in fact, refer to resources by proofs of membership in the list, constructed using 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 nth resource has the correct number of locks, later p is a proof that the $(n+1)$ resource has the correct number of locks. We have left two **holes** in this program, $\Box_1$ and $\Box_2$. Since dependently typed programs often contain proof terms, it can be convenient (and aid readability) to leave gaps for those proof terms. The type-checker will infer the required type for each hole, which the programmer is required to 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:

```
count : Nat → 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)))
```

Figure 4. Counting example

```
□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 it does not violate any priority orderings to lock the second resource. This is solved by isLater isFirst, which is valid because the first resource is unlocked. The second hole requires a corresponding proof for the first resource, under the assumption that the second one is already 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 from or writing to the resource, type-checking will fail because these proofs will not unify with the function type. If it does type-check, however, we can safely execute the program in multiple threads:

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

Note that this program uses the host language in several places. In particular, the **count** function takes two parameters that were passed using the host language. To execute this program, we need to construct an initial environment containing the relevant 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))
```

We can then run the program by calling the interpreter with this initial environment.

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

Efficiency

By introducing the intermediate interpreter stage, which eventually executes the low level instructions, we may seem to have introduced some abstraction overhead. We do not yet have an optimising compiler for IDRIS — our implementation is an interpreter that has been written in Haskell — and we therefore do not yet have realistic performance results. However, we believe that a compiled implementation of our approach should be competitive with a direct implementation: since **interp** is a total function, and programs such as **runThreads** and **count** above are known at compile-time, the interpreter can be *partially evaluated* against the input program. "Staging" an interpreter in this way, resulting in a translator to the host language, is a well-known technique [11, 46] that applies equally to dependently typed languages [7, 39], as to the conventionally typed languages where it has so far been applied.

6.7. 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 meta-theoretic 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 of a DSEL program corresponds to the specification given by the Lang typing rules, and that many of the required properties therefore follow directly from the definitions that we have given. We do not need separate proofs for these properties because the implementation *is* the proof. It therefore remains to show that the DSEL representation satisfies the required domain-specific properties.

Responsibilities and Assumptions

We begin by identifying the responsibilities of each language layer (i.e. IDRIS and the DSEL) and the DSEL programmer:

IDRIS as the host language, is responsible for providing the necessary low level operations, including forking, locking and creating references to shared memory. This is under the assumption that the low level implementation (we have used POSIX threads [9]) is correct. We have also assumed that the underlying type theory satisfies important meta-theoretic properties of other type theories such as strong normalisation, subject reduction and uniqueness of types [14].

The DSEL, along with its interpreter, is responsible for managing the low level operations safely with respect to a given set of resources. This is under the assumption that all of the resources passed to it are distinct.

The Programmer is responsible for constructing the set of resources used by a DSEL program, ensuring that all locks and references are distinct, and for fulfilling any proof obligations necessary for a program to be well-typed. This set of resources may be either static or dynamic, but in either case remains constant

throughout the DSEL program's execution. Additionally, in the current system, it is the programmer's responsibility to use the DSEL interface to concurrency operations rather than the low level interface.
Remark: We have made the assumption that all IORefs and Locks are distinct in the vector of resources passed to the interpreter. The DSEL *representation*, however, is independent of its *implementation*, therefore an alternative implementation of the interpreter could require a *proof* that resources are distinct. This would not change the form of any DSEL programs, but would change the way they were invoked.

Domain-Specific Properties

Programs in our DSEL satisfy two domain-specific properties, by construction, due to the design of the DSEL: 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 property 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 property is satisfied by enforcing a priority ordering on resources which eliminates the possibility of cyclic chains of resource requests [10]. The absence of deadlock in particular is hard to prove, because it requires dealing with all of the possible execution orders of systems of concurrent threads. By eliminating this necessary condition for deadlock, we obviate the need for any further formal statement of reachability or responsiveness of concurrent systems.

General Properties

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

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

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

- 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 type-checker will ensure that they remain consistent with each other. Correctness therefore relies on the specification of the DSEL itself being strong enough to express the required domain-specific properties.

6.8. The Next 700 Domain-Specific Embedded Languages?

We have described the implementation of our DSEL in two stages, namely: i) implementing a generic set of control constructs; and ii) 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/heap usage), it is desirable to make this separation explicit in the code. An alternative approach to adding constructors to our initial DSEL, would therefore be to parameterise it over the state-handling fragment of the language, and to add a new constructor which lifts the sub-language into the main language:

```
data Lang : (L : StateIn' → StateOut' → Ty) → StateIn → StateOut → Ty → ⋆ where
  lift : (prog : L s_in s_out T) → Lang L s_in s_out T
  | ...
```

We can then define the resource language independently:

```
data RLang : Vect ResState tin → Vect ResState tout → Ty → ⋆ where
    READ : (locked : ElemIs i (RState (S k) ty) tins) →
           RLang tins tins ty
  | WRITE : (val : interpTy ty) →
            (locked : ElemIs i (RState (S k) ty) tins) →
            RLang tins tins TyUnit
    ...
```

The interpreter for Lang is similarly parameterised over an interpreter for RLang. This follows Landin [27] 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 sub-language. Although we have not yet explored this, we anticipate that further development of this approach will also allow the composition of DSELs from smaller component DSELs.

7. Bank Accounts Revisited

In the example we gave in Section 6.6, the resources that we used were statically known. In many cases, this will be sufficient. For example, in an embedded system that manages shared access to specific hardware components, all the available resources will be known in advance of execution. What happens, however, 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 that is available in that account:

```
data AccountData : ⋆ where
  MkAcc : Int → AccountData
```

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

```
accounts : (n : ℕ) → Vect ResState n
accounts 0     ↦ nil
accounts (S k) ↦ cons (RState 0 (TyLift AccountData)) (accounts k)
```

```
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);
}
```

```
moveMoney : Int → Fin n → 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 recvEl
                 UNLOCK sendEl)
```

Figure 5. Money transfer pseudo-code with DSEL implementation

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). We 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 will distribute equally among the accounts:

```
mkAccounts : (n : ℕ) → IO (REnv (accounts n))
mkAccounts 0     ↦ return Empty
mkAccounts (s k) ↦ do kaccs ← mkAccounts k
                      aref ← newIORef (MkAcc 10000)
                      alock ← newLock 1
                      return (Extend (resource aref alock) kaccs)
```

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

7.1. First attempt

Figure 5 gives the pseudo-code for a money transfer, and shows how we implement that pseudo-code directly in the DSEL. Our dependent type system will reject this program for two reasons: 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 respect the priority ordering which the type system requires to guarantee absence of deadlock. In the remainder of this paper, we will see how the DSEL can be used to write a *correct* implementation of **moveMoney** and similar programs.

Remark: 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 its expansion instead.

The problems that we 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 then 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 $\Box_2$ is impossible to fill in, since we cannot guarantee that the sender, being locked first, is always the highest priority resource. $\Box_1$ is fine, because it is always possible to lock a resource when everything is unlocked.

The type system cannot identify whether the types of $\Box_1$ and $\Box_2$ are inhabited. It is therefore the programmer's responsibility to identify why a given program satisfies the required properties. The first problem is easy to fix — we can simply lift the unlocking outside the CHECK. To solve the second problem, which could cause deadlock, we consider the ordering of resources in order to justify the presence of a valid priority ordering to the DSEL.

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, then it is safe to lock i. In this way, we can construct a chain of priority-ordering proofs. First, we define LTFin, which 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 then 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)
```

Since resources could be used in any order at run-time, 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
```

Finally, 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, 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 (ISmall 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 higher 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 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 can 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:

$$\begin{aligned}&\mathbf{moveMoney} \;:\; \mathsf{Int} \to (sender \;:\; \mathsf{Fin}\; n) \to (receiver \;:\; \mathsf{Fin}\; n) \to \\ &\qquad\qquad \mathsf{Lang}\; (\mathbf{accounts}\; n)\; (\mathbf{accounts}\; n)\; \mathsf{TyUnit}\\ &\mathbf{moveMoney}\; sum\; s\; r\\ &\quad \mapsto \mathbf{moveMoney'}\; sum\; s\; r\; (\mathbf{cmpFin}\; s\; r)\end{aligned}$$

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

Remark: The program here is fairly simple, and the proof of lock safety more tricky. It is worth noting, however, that the complexity of the proofs is a function of the complexity of the *lock structure* rather than of the program itself. So however complex a function that manipulates two accounts, the complexity of the proofs will not increase. Conversely, a program that manipulates more than two accounts simultaneously, or interleaves locking/unlocking of multiple accounts, will require more reasoning. Additionally, we note (as described in Section 3.2) that none of the proofs are retained at run-time, so the presence of proof terms in the program does not affect the efficiency of execution in any negative way. In some cases, we may even find the presence of proof terms has a positive effect, in that some run-time checking which would otherwise be necessary may be proved to be unnecessary, and so discarded.

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 IDRIS. The body of the function is the same in each case, reading and writing the sum of money, but the lock ordering and the types of the required proofs are different. We can, of course, lift out the body into a common polymorphic auxiliary function that is parameterised 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. Unfortunately, since full type inference for dependently typed languages is not possible, lifting the body into a separate function means that the *programmer*, rather than the type-checker, is required to work out the intermediate type. In this kind of situation, we would expect that, in a more mature programming language, the type-checker would be able to perform some limited type inference on auxiliary functions.

8. Related work

Reasoning about locks and shared memory is an instance of a more general problem of resource usage verification, where resources could include files, heap usage, running time, power consumption, among many others. 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* [32] 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*, but 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, as long as the relevant proof terms can be constructed. Various special-purpose type systems have also been developed. For example, [48] can be used to enforce security properties and [25, 44] can be used to prevent deadlock and race conditions. Igarashi and Kobayashi have also developed a type system [22] for guaranteeing resource properties. While these approaches seem promising as specialised applications, we prefer to build on a strong general-purpose type system, thereby avoiding the need to develop new soundness proofs and a new type-checking implementation. By separating the generic and specific components of a DSEL, as described in Section 6.8, our approach allows the construction of *composable* domain-specific embedded languages. However, the generality and precision of our approach comes at a price: DSEL programs require embedded proofs which must be provided by the programmer.

An alternative approach involves exploiting program monitoring [29] to *dynamically* check that a program adheres to security constraints, and to take remedial action before a dangerous action is executed. This has the advantage that no changes or annotations are needed, and there is no requirement to construct a priority ordering, but the disadvantage that checks are made at run-time so no guarantee can be made that a program will execute successfully. Our approach differs 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. Again, this is at the cost of requiring explicit proof terms.

An obvious approach to dealing with resources is to use linear types, e.g. [18, 19]. Linear types are ideal for capturing many resources properties because the type system enforces that a value can be used only once, and indeed linear types have been used to analyse communication in concurrent programming languages [21, 26], using an alternative approach which guarantees that concurrent channels are used once. We have avoided linear types primarily because we believe that dependent types are sufficiently strong to deal with the explicit management of resources within a uniform type system, but also because linearity itself does not capture the priority ordering, so an alternative model of concurrent resources is needed. The approach we have taken reflects common practice among concurrency practitioners [38, 45], in which a priority ordering for resources is constructed and resource accesses are dynamically checked.

There are several other approaches to concurrent programming which are not based on a traditional view of locks on shared resources, and so do not require a potentially arbitrary priority ordering to be set up. For example, software transactional memory [17] works in terms of *transactions*, without regard to what other threads may be doing, with a final *commit* operation which validates and applies transactions. While this has benefits for correctness, there are still efficiency problems being addressed [13, 31]. Erlang[4] conversely follows the actor model [1] in which processes send messages rather than share data.

Like our work, YNot [37] aims to reason about side-effects in imperative, in this case using dependent types in Coq. In this system, imperative programs are annotated with Hoare Logic style formal pre- and post-conditions. A related approach using Hoare Type Theory to reason about transaction based concurrency is presented in [35]. In contrast, we prefer to add the minimal required imperative features to functional programs, and the DSEL approach allows us to focus on domain-specific constraints. 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) [40] in Haskell or Omega [43], or perhaps even by enforcing resource constraints using Haskell type classes, as suggested by Kiselyov and Shan [24]. 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.

[4] `http://www.erlang.org/`

9. Conclusion

Concurrent programming is becoming increasingly important, as multi-core architectures become increasingly common. As a result, it is critical that programming languages provide sufficient support to allow programmers to take full advantage of the opportunities for concurrent program execution. We have shown how to develop Domain-Specific Embedded Languages in a dependently typed meta-language, IDRIS. These DSELs 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 directly derive some important safety properties of programs written in the concurrent DSEL:

- 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 new 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 also identify some weaknesses in current tools. In particular, code reusability can be difficult. Generic code which is syntactically the same in more than one place, such as the body of each branch of $\mathbf{moveMoney}'$, can nevertheless have different types in each place. 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, e.g. following the bidirectional type-checking ideas of [30]. Furthermore, we often find ourselves implementing the same data structure several times, with different indices, e.g. Vect and Env are both implemented as lists. It would be preferable to implement functions over such structures once, generically. Nevertheless, if we do want total static safety, we cannot 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 worrying about detailed correctness proofs. By defining the required preconditions in advance, in the definition of the DSEL data type, the type checker informs a programmer of the properties they need to show to complete the correctness proof. 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 proofs are straightforward, but where resources are dynamic, as in our bank account example, some simple reasoning is required. The DSEL author can help this reasoning by providing a library of useful lemmas, such as $\mathbf{lockEarlier}$ and a generic version of $\mathbf{unlockedAcc}$. In all cases, the application programmer is able to construct such ordering proofs dynamically — the typing rules will still ensure that deadlock does not occur, since they will ensure that any necessary checks are executed.

9.1. Future work

There are a number of areas that would repay further study, relating to both the design and implementation of DSELs and the host language IDRIS itself. 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 limiting the lock count to one), with semaphores, or with shared variables. We would expect to be able to handle the latter by extending the type of FORK, but would then also need to deal with inter-thread synchronisation and communication. 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 operating-system and device-driver programmers to exploit modern processors while using safe abstractions. 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 [3]. In this context, one especially interesting empirical measure of the value of the approach would be whether correct-by-construction programs could detect errors in existing protocols. Fourthly, one important resource usage problem we have previously looked at using other methods [6], 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, since, 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 a promising way to address this problem, however, because the BIND operation provides a mechanism to track the changes in the resource state at each stage of the program's execution. Finally, the main weakness in our approach is the requirement on the programmer to provide proof terms to guarantee that locks are taken in priority order. While these proof terms are generally easy to construct in practice, it would be preferable for them to be constructed automatically by a (possibly partial) decision procedure where possible. We believe that for our approach to be usable in practice, and for resource analysis in general, domain-specific embedded languages will require domain-specific decision procedures.

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] S. Bhatti, E. Brady, K. Hammond, and J. McKinna. Domain specific languages (DSLs) for network protocols. In *International Workshop on Next Generation Network Architecture (NGNA 2009)*, 2009.

[4] E. Brady. *Practical Implementation of a Dependently Typed Functional Programming Language*. PhD thesis, University of Durham, 2005.

[5] E. Brady. Ivor, a proof engine. In *Implementation and Application of Functional Languages 2006*, volume 4449 of *LNCS*. Springer, 2007.

[6] 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.

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

[8] 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.

[9] D. R. Butenhof. *Programming with POSIX Threads*. Addison Wesley, 1997.

[10] E. Coffman, M. Elphick, and A. Shoshani. System deadlocks. *ACM Computing Surveys*, 3(2):67–78, 1971.

[11] K. Czarnecki, J. O'Donnell, J. Striegnitz, and W. Taha. DSL implementation in MetaOCaml, Template Haskell, and C++. In *Domain Specific Program Genearation 2004*, volume 3016 of *LNCS*. Springer, 2004.

[12] C. Elliott and P. Hudak. Functional reactive animation. In *International Conference on Functional Programming*, pages 163–173, June 1997.

[13] R. Ennals. Software transactional memory should not be abstraction free. Technical Report IRC-TR-06-052, Intel Research, 2006.

[14] H. Goguen. *A Typed Operational Semantics for Type Theory*. PhD thesis, University of Edinburgh, 1994.

[15] 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.

[16] 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.

[17] 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.

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

[19] 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 La nguages*. ACM, 2003.

[20] P. Hudak. Building domain-specific embedded languages. *ACM Computing Surveys*, 28A(4), December 1996.

[21] A. Igarashi and N. Kobayashi. Type-based analysis of communication for concurrent programming languages. In *SAS '97: Proceedings of the 4th International Symposium on Static Analysis*, pages 187–201, London, UK, 1997. Springer-Verlag.

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

[23] 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.

[24] O. Kiselyov and C.-C. Shan. Lightweight static resources: Sexy types for embedded and systems programming. In *Draft Proc. Trends in Functional Programming (TFP '07)*, 2007.

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

[26] N. Kobayashi, B. C. Pierce, and D. N. Turner. Linearity and the pi-calculus. *ACM Trans. Program. Lang. Syst.*, 21(5):914–947, 1999.

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

[28] D. Leijen. Parsec, a fast combinator parser. `http://www.cs.uu.nl/~daan/parsec.html`, 2001.

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

[30] A. Löh, C. McBride, and W. Swierstra. A tutorial implementation of a dependently typed lambda calculus, 2009. To appear in Fundam. Inf.

[31] V. J. Marathe and M. Moir. Efficient nonblocking software transactional memory. In *PPoPP '07: Proceedings of the 12th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 136–137, New York, NY, USA, 2007. ACM.

[32] 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.

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

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

[35] A. Nanevski, P. Govereau, and G. Morrisett. Towards type-theoretic semantics for transactional concurrency. In *Workshop on Types in Language Design and Implementation (TLDI'09)*. ACM, 2009.

[36] A. Nanevski, G. Morisett, and L. Birkedal. Polymorphism and separation in Hoare Type Theory. In *Proc. 2006 International Conf. on Functional Programming (ICFP 2006)*, pages 62–73. ACM, 2006.

[37] A. Nanevski, G. Morrisett, A. Shinnar, P. Govereau, and L. Birkedal. Ynot: dependent types for imperative programs. In *International Conf. on Functional Programming (ICFP 2008)*, pages 229–240. ACM, 2008.

[38] S. Oaks and H. Wong. *Java Threads*. O'Reilly, 3rd edition, 2004.

[39] 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.

[40] 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.

[41] 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.

[42] 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.

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

[44] K. Suenaga and N. Kobayashi. Type-based analysis of deadlock for a concurrent calculus with interrupts. In *Proc. European Symposium On Programming (ESOP '07)*, 2007.

[45] H. Sutter. Use lock hierarchies to avoid deadlock. *Dr Dobb's*, December 2007.

[46] W. Taha. *Multi-stage Programming: Its Theory and Applications*. PhD thesis, Oregon Graduate Inst. of Science and Technology, 1999.

[47] 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.

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