


A Framework for Resource Dependent EDSLs in a Dependently Typed Language

Jan de Muijnck-Hughes 

University of Glasgow, United Kingdom

Jan.deMuijnck-Hughes@glasgow.ac.uk

Edwin Brady 

University of St Andrews, United Kingdom

ecb10@st-andrews.ac.uk

Wim Vanderbauwhede 

University of Glasgow, United Kingdom

Wim.Vanderbauwhede@glasgow.ac.uk

Abstract

Idris' Effects library demonstrates how to embed resource dependent algebraic effect handlers into a dependently typed host language, providing run-time and compile-time based reasoning on type-level resources. Building upon this work, RESOURCES is a framework for realising Embedded Domain Specific Languages (EDSLs) with type systems that contain domain specific substructural properties. Differing from Effects, RESOURCES allows a language's substructural properties to be encoded within type-level resources that are associated with language variables. Such an association allows for multiple effect instances to be reasoned about autonomously and without explicit type-level declaration. Type-level predicates are used as proof that the language's substructural properties hold. Several exemplar EDSLs are presented that illustrates our framework's operation and how dependent types provide correctness-by-construction guarantees that substructural properties of written programs hold.

2012 ACM Subject Classification Software and its engineering → General programming languages; Software and its engineering → Language features; Software and its engineering → Domain specific languages; Software and its engineering → System modeling languages

Keywords and phrases Dependent Types, Algebraic Effect Handlers, Domain-Specific Languages, Embedded Domain Specific Languages, Idris, Substructural Type-Systems

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2020.20

Category Pearl

Supplementary Material ECOOP 2020 Artifact Evaluation approved artifact available at <https://doi.org/10.4230/DARTS.6.2.2>.

Funding This work was funded by EPSRC projects: *Border Patrol: Improving Smart Device Security through Type-Aware Systems Design* (EP/N028201/1); and *Type-Driven Verification of Communicating Systems* – EP/N024222/1.

Acknowledgements The authors would like to thank the anonymous reviewers for their excellent reviews that served to better the work.

1 Introduction

Substructural Type-Systems allow type-systems to reason about abstract resources associated with the type-system's domain of operation [65]. For general purpose programming languages these resources typically capture, and reason quantitatively about, memory access, variable usage, and erasure of non-essential terms. However, not all languages are general purpose, nor are their abstract resources quantitative in nature cf. Linear Typing with Session



© Jan de Muijnck-Hughes, Edwin Brady, and Wim Vanderbauwhede; licensed under Creative Commons License CC-BY

34th European Conference on Object-Oriented Programming (ECOOP 2020).

Editors: Robert Hirschfeld and Tobias Pape; Article No. 20; pp. 20:1–20:31

Leibniz International Proceedings in Informatics



Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany



Types [30]. Domain Specific Languages (DSLs) are special purpose languages tailored to a specific application domain [26]. Embedded Domain Specific Languages (*EDSLs*) are DSLs that have been embedded within a host language to capitalise upon the host language’s functionality. Implementing an EDSL with a substructural type-system, however, requires an implementation language that not only supports substructural typing but supports reasoning about domain specific substructural properties.

Algebraic effect handlers support reasoning about a program’s side-effects [50] and several programming languages such as OCaml and Haskell have been extended with them [33, 46]. **Effects** [11] is a general purpose *resource dependent* algebraic effect handler library for the dependently typed programming language Idris [10]. Through **Effects**, developers can realise EDSLs with substructural type-systems.

Effects has been realised as *Resource Dependent EDSLs* in which the EDSL is specified as an algebraic data type whose type captures resources that are each associated with an abstract state machine [11]. The EDSL’s type forms a Hoare monad [1, 7] and sequencing of expressions captures valid transitions between individual states of these resources. Such EDSL construction is a common design pattern seen within dependently typed programming languages. For example, there are EDSLs for reasoning about: communicating systems [13, Chp. 15]; communication protocols [19]; and hardware component interfaces [20].

Effects requires, however, that domain specific effects operate within a general purpose effectful context, and effect management is not an autonomic aspect of the program and is the responsibility of the programmer. That is, effect instances describe a single effect within the program, and multiple same effect instances must be explicitly labelled. Figure 1 illustrates these issues with a simple copy function that opens two file handles and writes a single line from one file to another¹. Within the function’s body each same effect instance must be labelled at both the value and type-level. Use of **Effects** is not ideal when designing EDSLs with domain specific effect systems in which multiple same effect instances can occur, nor does the **Effects** library support autonomic effect management.

```
copy : (o, n : String) -> Eff (Maybe FileError) [A ::: FILE (), B ::: FILE (), STDIO]
copy o n = do
  Success <- A :- open o Read | FError e => do {println e; pure (Just e)}
  Result s <- A :- readLine | FError e => do {println e; A :- close; pure (Just e)}
  A :- close
  Success <- B :- open n WriteTruncate | FError e => do {println e; pure (Just e)}
  res <- B :- writeString s
  case res of
    Success => do {B :- close; pure Nothing}
    FError e => do {println e; B :- close; pure (Just e)}
```

■ **Figure 1** Example of labelled effects using Idris’ **Effects** library.

1.1 Contributions

We build on previous work in designing algebraic effect handlers in Idris [11, 10]. Rather than associating an effect’s abstract resource with the program itself we associate it with a bound variable within the EDSL. Further, the list of possible effects is now constrained to an *a priori* set of domain specific effects. Such an association and restriction leads to greater reasoning and manipulation of the effects within a Resource-Dependent EDSLs, thus enabling autonomic effect management and reasoning about the state of an effect’s resource. Given this principal idea, our contributions are:

¹ Idris’ *pattern match & bind* notation reduces the number of case expressions required [11]. This notation supports binding to a value and presentation of the remaining cases on the right.

1. **RESOURCES**, a general purpose framework for constructing Resource Dependent EDSLs that have a domain specific substructural type-systems. Further, we illustrate using *effect handlers* how EDSLs created using **RESOURCES** can be operated on in a variety of different evaluation contexts.
2. A collection of exemplar EDSLs demonstrating the ability of **RESOURCES** to create EDSLs. **Files** reasons about multiple concurrent File IO (Section 4.1); **Wireless** reasons about domain specific bigraph construction (Section 4.2); and **Sessions** captures value dependent global session descriptions – Section 4.3.

RESOURCES is a step forward for developers and presents a new general framework for realising domain specific substructural type-systems for resource dependent EDSLs. Thus, supporting the exploration of novel type-systems similar to those seen in existing systems [47, 28, 17, 32].

1.2 Outline

Section 2 discusses how dependent types support type-level abstract state machines, and reasoning about such machines. Section 3 presents the framework itself, and exemplar EDSLs appear in Section 4. Sections 2 and 3 describes how data types modelled after *De Bruijn indices* [18] provide type-level assertions that certain substructural properties hold.

► **Remark.** Although not essential, before reading about our work we encourage readers not familiar with Idris to learn more about the language, its syntactic constructs, auto-implicit arguments, and semantic highlighting².

2 Type-Level State Tracking and Reasoning

This section introduces the underlying technique for type-level reasoning about abstract resources, through implementation of an EDSL that captures high-level file interactions.

Within our EDSL files are either: closed; open for reading; or open for writing. We encapsulate these operations using the following four operations, and a helper function for displaying showable data: **Open** – which opens a file for reading or writing; **Read** – which reads a string from a file opened for reading; **Write** – which writes a string to a file opened for writing; **Close** – which closes an already open file; and **PrintLn** – which prints showable data.

Parameterised monads allow for language expressions to be associated with a type-level state which we refer to as a *resource* [1]. Hoare monads allow for state transitions to be presented at the type-level [7]. The type of each expression describes how the expressions affects the abstract state. An operation and its type give a Hoare Triple [29]. Definition of state machines within such a monadic construct ensures that any sequence of operations which type checks is a valid sequence of operations. For our example, this means that any operation on a file must respect the type-level state machine we define. Thus, attempting to write to a file opened for reading should present itself as a type error.

Figure 2 presents an implementation of the EDSL within a Hoare monad. **FileIO** is parameterised by the state of the file *before* and *after* each operation. The arguments to **FileIO** are: a **Type**, which represents the return type of the operation; and two **FileState**

² Differing from Idris’ existing colouring scheme, we use a more printer friendly set: **Data** constructors; **Type** constructors; **Bound** variables; named **Function**; Idris **Keywords**; and *Implicitly* bound variables. Agda style highlighting is used for **typed holes**.

20:4 A Framework for Resource Dependent EDSLs in a Dependently Typed Language

instances, which represent the *input* state (the precondition) and the *output* state – the postcondition. These invariants ensure that read and write operations only work when the state of the file is correct: Open for their particular mode of operation. The `PrintLn` operation should not affect the program’s abstract state. The type of `Bind` explains how sequencing changes the file’s state based on a previous expression. `Pure` returns a pure value. Thus, if a sequence of `FileIO` expressions type checks, then it is a valid sequence of operations according to the stated protocol. Rather than use `Bind` and `Pure` directly, `do`-notation is realised by overloading `(>=)` and `pure`, with `Bind` and `Pure`.

```
data Mode = R | W

data State = Open Mode | Closed

data FileIO : (type : Type) -> (pre : State) -> (post : State) -> Type where
  Bind : FileIO a stA stB -> a -> FileIO b stB stC -> FileIO b stA stC
  Pure : a -> FileIO a before after

  Open  : (fname : String) -> (m : Mode) -> FileIO () Closed (Open m)
  Read  : FileIO String (Open R) (Open R)
  Write : (value : String) -> FileIO () (Open W) (Open W)
  Close : FileIO () (Open m) Closed
  PrintLn : Show a => a -> FileIO () curr curr
```

■ **Figure 2** An EDSL for interacting with a single file.

Figure 3 presents a sample program written in `FileIO`. The program’s abstract state is initialised to `Closed`. Each expression transitions the state according to the rules embedded in the type of our EDSL. If an incorrect sequence of expressions were to be given, for example opening two files or reading to a file opened for writing, then the program would fail to type-check.

```
toFile : (fname : String) -> (contents : String) -> FileIO () Closed Closed
toFile fname str = do { Open fname W; Write str; Close }
```

■ **Figure 3** An example program for interacting with a single file.

2.1 Files with Errors

The definition for `FileIO` is not sufficiently expressive: Operations on file handles are naturally impure; `FileIO` is pure. The EDSL does not capture potential errors that occur when interacting with a file. For example, being unable to open a file handle, or an error occurring during a read/write operation.

Figure 4a illustrates how the type of `FileIO` can be redefined to address run-time errors. The post-condition is now a function that computes the resulting state dependent on the value returned by the expression. For example, `Open` changes the state to `Closed` in the result of an error, otherwise the state remains the same. The remaining constructors for `FileIOE` can be redefined accordingly. Figure 4b shows Figure 3 rewritten using `FileIOE`. If the result of `Open` or `Write` is not checked, the subsequent interactions will not type check. The next state would be unknown.

```

data FileIOE : (type : Type) -> (pre : State) -> (post : type -> State) -> Type where
  Open : (fname : String)
        -> (m : Mode)
        -> FileIOE (Maybe FileError)
            Closed
            (\res => case res of {Nothing => Open m; Just err => Closed})
  ...

```

(a) Partial redefinition of FileIO.

```

toFile : String -> String -> FileIOE (Maybe FileError) Closed (const Closed)
toFile fname str = do
  Nothing <- Open fname W          | Just err => do {PrintLn err; pure err}
  Nothing <- Write fh "A string" | Just err => do {PrintLn err; Close fh; pure err}
  Close fh
  pure Nothing

```

(b) Figure 3 rewritten as an FileIOE instance.

■ **Figure 4** Redefining FileIO to include type-level enforcement of error handling.

This pattern of state-aware EDSL construction allows reasoning about the abstract state of an EDSL at *compile-time*, based on data obtained at *run-time*. However, FileIOE is not expressive enough to reason about, nor interact with, multiple files.

2.2 Modelling Multiple File Access with Errors

Figure 5 extends the definition of FilesIOE with a list of abstract state machines: one per open file.

```

data FilesIOE : (ty : Type) -> (old : List Item) -> (new : ty -> List Item) -> Type where
  Pure : (val : a) -> FilesIOE a (st val) st

  Bind : FilesIOE a first snd_fn
        -> ((x : a) -> FilesIOE b (snd_fn x) third_fn)
        -> FilesIOE b first third_fn

  Open : (fname : String) -> (m : FMode)
        -> FilesIOE (Either FileError Handle) old
            (\res => case res of {Right hdl => MkItem hdl (Open m)::old; Left _ => old})

  Read : (hdl : Handle) -> (prf : Any (IsOpenFor hdl R) item old)
        -> FilesIOE (Either FileHandle String) old
            (\res => case res of {Right _ => old; Left _ => update (closeHandle) old prf})

  Write : (hdl : Handle) -> (str : String) -> (prf : Any (IsOpenFor hdl W) item old)
        -> FilesIOE (Maybe FileError) old
            (\res => case res of {Nothing => old; Just _ => update (closeHandle) old prf})

  Close : (hdl : Handle) -> (prf : Any (IsHandle hdl) item old)
        -> FilesIOE () old (const $ drop old prf)

  PrintLn : Show a => (msg : a) -> FilesIOE () old (const old)

```

■ **Figure 5** An EDSL to model multiple concurrent file interactions.

To help with reasoning about multiple files we introduce two helper data structures.

```

data Handle = MkHandle
data Item = MkItem Handle FileState

```

Handle represents file handles at both the value and type level, and Item associates a type-level file state with a particular handle. File handles are bound to names using the Bind constructor. Although, we could use a nameless representation based on *De Bruijn* indices

we can take advantage of Idris' elaborator to distinguish between different instances of `Handle` based on their bounded names and type-level values. During the type checking process Idris' elaborator translates high level Idris code to the internal type theory representation [9] by expanding high level language constructs such as case blocks and where clauses, and inferring values for implicit arguments by unification and search. Further, by replacing the original state that parameterises `FileIOE` with a list of these `Item` instances, the state of each open file handle in our EDSL can be tracked.

Figure 6 presents two type-level *predicates* for reasoning about individual items in our type-level context. `IsOpenFor` declares that the given file handle has been opened for reading or writing; and `IsHandle` declares that the given file handle exists. To aid reasoning about multiple file handles, i.e. any item in the program's context, the list quantifier `Any` is used. The `Any` represents existential quantification that the given predicate holds on a list item.

With Idris' do-notation let-bindings are provided, however, let-bindings do not interact with the type-level context. We can use the `Any` list quantifier in conjunction with Idris' elaborator to ensure that aliased variables cannot be used. If an operation with an aliased handle were to be used then proof (witness) cannot be given of the handle's existence in the type-level context as Idris' elaborator will fail to associate the aliased named with an abstract state.

```
data IsOpenFor : (hdl : Handle) -> (mode : FMode) -> (item : Item) -> Type where
  FileIsOpenFor : (m : FMode) -> IsOpenFor hdl m (MkItem hdl (Open m))
```

(a) Predicate for reasoning about file handle mode.

```
data IsHandle : (hdl : Handle) -> (item : Item) -> Type where
  FileExists : (hdl : Handle) -> IsHandle hdl (MkItem hdl st)
```

(b) Predicates for linking file handle to instance of `Handle`.

■ **Figure 6** Predicates.

Figure 7 presents the type signatures for two helper functions that manipulate the type-level context based on an expression's associated predicates. The first function, `update`, updates specific elements in our context dependent upon the supplied `Any` proof. Further, the update function `f` facilitates access to the predicate that holds over the item we are updating. The `drop` function removes an item from the context using the supplied `Any` proof about the item.

```
update : (f : (i : Item)
          -> (prf : p i)
          -> Item)
        -> (context : List Item)
        -> (index : Any p item context)
        -> List Item
```

(a) Updating an `Item` instance.

```
drop : (context : List Item)
      -> (index : Any p item context)
      -> List Item
```

(b) Removing an `Item` instance.

■ **Figure 7** Functions for manipulating the type-level context.

With these extra data structures, and predicates, type-level operations on individual file handles in `FilesIOE` becomes autonomic. Files are opened using `Open` which extends the context with the new file handle, its initial state. The old context is retained if the operation fails. Reading a file, using `Read`, requires proof that the file is already open for reading. We do so using `IsOpenFor` and `Any`. If the read is successful then the old context is retained. If the read is unsuccessful then the state of the file is updated to `IsClosed`. The definition of `Write` is analogous to `Read`. Closing a file (`Close`) removes the file's state from the context. For a close operation to be allowed, evidence must be presented that the file *is* in the closed state. With this evidence the file's associated state can be removed.

Figure 8a presents an example of a program written using `FilesIOE`. For each language expression that requires a predicate a proof must also be given. With this approach value level expressions become incredibly verbose. This is *too* verbose. Users should not be expected to write such proofs by hand. Idris supports *auto-implicit* arguments, in which the values for implicit arguments to a function can be automatically constructed using a greedy constructor-based search to find a value that matches the arguments type. By wrapping each language expression in a function that uses auto implicits we can automatically construct the proofs.

```
copy : (old, new : String) -> FilesIOE (Maybe FileError) Nil (const Nil)
copy old new = do
  Right fh <- Open old R | Left err => do {PrintLn err; Pure (Just err)}
  Right str <- Read fh (H $ FileIsOpenFor R) | Left err => do
    PrintLn err
    Close fh (H $ FileExists fh)
    Pure (Just err)
  Close fh (H $ FileExists fh)
  Right fh1 <- Open new W | Left err => do {PrintLn err; Pure (Just err)}
  res <- Write fh1 str (H $ FileIsOpenFor W)
  case res of
    Nothing => do {Close fh1 (H $ FileExists fh); Pure (Nothing)}
    Just err => do {PrintLn err; Close fh1 (H $ FileExists fh1); Pure (Just err)}
```

(a) With Proofs.

```
copy : (old, new : String) -> FilesIOE (Maybe FileError) Nil (const Nil)
copy old new = do
  Right fh <- open old R | Left err => do {println err; pure (Just err)}
  Right str <- read fh | Left err => do
    println err
    close fh
    pure (Just err)
  close fh
  Right fh1 <- open new W | Left err => do {println err; pure (Just err)}
  res <- write fh1 str
  case res of
    Nothing => do {close fh1; pure (Nothing)}
    Just err => do {println err; close fh1; pure (Just err)}
```

(b) With Proofs calculated using auto-implicit arguments.

■ **Figure 8** Figure 3 rewritten using `FilesIOE`.

For example, the wrapper function for `Close` would be written as:

```
close : (h : Handle) -> {auto idx : Any (IsHandle h) i o}
  -> FileIOE () o (const (drop o idx))
close h {idx} = Close h idx
```

By convention, the function that calculates an auto-implicit argument is named using lower case variants of the constructor name. Figure 8b presents the “cleaned” version of Figure 8a.

Notice that for each branch in our case-splits, and bind operations, we must close open file handles. Here the type-level state requires us to exit functions with an empty context. This ensures that all file handles are closed when we exit our program. To ensure that our programs start and end with the correct states the type-synonym `FilesIOE` is defined to ensure that the end state of the program must be empty, implying that all file handles that were open, were also closed.

```
FileIO : Type -> Type
FileIO ty = FilesIOE ty Nil (const Nil)
```

3 The Framework

The definition of `FilesIOE` follows a pattern of EDSL construction seen in existing work [13, 19, 20]. This section describes the implementation of `RESOURCES` that encapsulates the common structures and definition common to these EDSLs.

3.1 Capturing Abstract State

Central to the framework's operation is associating variables with an abstract state that is reasoned about at the type-level. Figure 9 presents the definitions for variables, their associated state, and an EDSLs' context. The relationship between a variable and a state is captured by indexing the type for variables (`Var`) and state items (`StateItem`) with a data type that acts as a meta-type representing the type of the variable's associated state. Following from the *Well-Typed Interpreter* [3], the type of `StateItem` is further indexed by a function to compute the concrete type associated with the type-level value. The definition of `StateItem` associates an instance of `Var` with a specific instance of state. As we saw in Section 2.2, Idris' elaborator allows us to distinguish between different instances of `Var`. The list of state items captured at the type level, the EDSLs context, is collected in a bespoke data type `Context`.

```

data Var : (Ty : Type) -> (ty : Ty) -> Type where
  MkVar : Var type value

data StateItem : (ty : Type) -> (calcSTy : ty -> Type) -> (value : ty) -> Type where
  MkStateItem : (value : type)
    -> (label : Var type value)
    -> (state : calcSTy value)
    -> StateItem type calcSTy value

data Context : (type : Type) -> (calcSTy : type -> Type) -> Type where
  Nil : Context type calcSTy
  (::) : (item : StateItem type calcSTy value)
    -> (rest : Context type calcSTy)
    -> Context type calcSTy

```

■ **Figure 9** Definitions for variables, state items, and type-level context.

3.2 Sequencing Language Expressions

Figure 10 presents the parameterised data type that captures state transitions between different abstract states. A type-synonym ensures that all languages defined using the framework use the same signature. A language expression has an expression type (`exprTy`), an existing `Context` instance `pre`, and a function `postK` to compute the new context from the expression's value. `Lang` is a function that constructs an instance of this type signature with the meta-type and a function to compute concrete states indexing the signature.

```

Lang : (type : Type) -> (type -> Type) -> Type
Lang type calcSTy = (exprTy : Type)
  -> (pre : Context type calcSTy)
  -> (postK : exprTy -> Context type calcSTy)
  -> Type

```

■ **Figure 10** The type for all EDSLs.

Figure 11 presents `LANG`, a single data type, to collate the: meta-type – (`type`); interpreter (`calcSTy`); and `Lang` instance together.

```

data LANG : Type -> (type : Type) -> (calcSTy : type -> Type) -> Type where
  MkLang : (type : Type)
    -> (calcSTy : type -> Type)
    -> Lang type calcSTy
    -> LANG type calcSTy

```

■ **Figure 11** Data Structure and accessors to hold EDSL Specifications.

Figure 12 presents `LangM`, the data structure that captures, generically, the sequencing of EDSL language expressions. The data type `LangM` removes the need for each EDSL to provide the same definitions for sequencing expressions. The constructor `Value` returns a pure value. `Let` provides sequencing of expressions and insertion of computed values into subsequent expressions. `Expr` provides embedding of EDSL language expressions into `LangM`. The type of `LangM` is indexed by: `m` – a monadic context; `exprTy` – the type associated with an expression; `spec` – the language specification that is being sequenced; `pre` – the original context; and `postK` – the computed context.

```

data LangM : (m : Type -> Type)
  -> (exprTy : Type)
  -> (spec : LANG type calcSTy)
  -> (pre : Context type calcSTy)
  -> (postK : exprTy -> Context type calcSTy)
  -> Type where
  Value : (value : a) -> LangM m a spec (postK value) postK

  Let : LangM m a spec old oldK
    -> ((val : a) -> LangM m b spec (oldK val) postK)
    -> LangM m b spec old postK

  Expr : {eSig : Lang type calcSTy}
    -> (expr : eSig a pre postK)
    -> LangM m a (MkLang type calcSTy eSig) pre postK

```

■ **Figure 12** Definition of `LangM`.

3.3 Reasoning About Abstract State

Within dependently typed languages, list quantifiers such as `All` and `Any` are based on *De Bruijn* indices and reasoning about all or specific elements within a standard list using a provided predicate [18]. Figure 13 presents similar predicated quantifiers that can be constructed for `Context`. The `AllContext` predicate mirrors `All` and allows one to present a predicate that applies to all state items. Mirroring `Any`, `InContext` constructs a proof that there is an element (searching from the head of the list) satisfying the provided predicate.

```

data AllContext : (p : (value : type) -> (item : StateItem type calcSTy value) -> Type)
  -> (c : Context type calcSTy)
  -> Type where

data InContext : (value : type)
  -> (p : StateItem type calcSTy value -> Type)
  -> (c : Context type calcSTy)
  -> Type where

```

■ **Figure 13** Quantifiers for reasoning about elements in `Context`.

Generic functions can be constructed using these quantifiers to operate on `Context` instances. Specifically, instances of `InContext` provide type-safe transformations on specific elements. Figure 14 presents the definition of several of these functions. The first function, `update`, updates specific elements in our context dependent upon the supplied `InContext` proof. Here, the update function `f` facilitates access to the predicate that holds over the item being updated. The function `drop` removes an item from the context using the `InContext` proof about the item. A third function `setState` allows the state to be replaced.

```

update : (context : Context type calcSTy)
  -> (index   : InContext value predicate context)
  -> (f : (item : StateItem type calcSTy value)
    -> (prf  : predicate item)
    -> StateItem type calcSTy value)
  -> Context type calcSTy

drop : {predicate : StateItem type calcSTy value -> Type}
  -> (context   : Context type calcSTy)
  -> (index     : InContext value predicate context)
  -> Context type calcSTy

setState : {predicate : StateItem type calcSTy value -> Type}
  -> (context   : Context type calcSTy)
  -> (index     : InContext value predicate context)
  -> (item'    : calcSTy value)
  -> Context type calcSTy

```

■ **Figure 14** Functions acting on `Context` instances.

3.4 Language Evaluation

The `Effects` library uses `Idris` interfaces to link effect specifications (descriptions) to implementation handlers that realise the specification for a specific implementation context. This is the `Handler` interface. Figure 15 presents a similarly named interface to describe EDSL evaluation and effect handling, together with a secondary interface, `RealVar` that details how variables in an EDSL are to be translated to concrete types. Within `RESOURCES` our individual effect specifications will be subterms in our EDSL and their handlers the corresponding body in the implementation.

The `Handler` interface is indexed by: the meta-type type; the meta-type interpreter; a language expression specification; an accumulator; and a specific evaluation context. Similarly to the `Effects` handler interface, instances of `Handler` detail how to evaluate EDSL expressions in a specific evaluation context, and how the domain specific effects are to be handled. The function `handle` takes an evaluation environment, the expression to be considered, an accumulator, and a continuation to pass on the updated environment and accumulator.

Figure 16 presents the definition (`Env`) for evaluation environments to keep track of variables and their abstract state. The type of `Env` is indexed by an evaluation context `m` and the current state of the EDSL (`ctxt`) during evaluation. This ensures that the items in the environment grows and shrinks as the type-level context (`ctxt`) grows and shrinks. The data type `Tag` is a container for holding concrete variable representations. The function of the `RealVar` interface computes the concrete type from the language’s meta-type.

Section 3.3 presented predicates for reasoning about state items in instances of `Context`. These same predicates are used to provide operations on our computation environments; `Env` is indexed by a context. Figure 17 presents the function definitions for `lookup`, `update`, and `drop` that mirror the functions presented in Section 3.3. When specifying how EDSLs are evaluated, type-level operations on the context *must* be mirrored at the value level for the environment.

```

interface RealVar (type : Type) where
  CalcRealType : type -> Type

interface RealVar type
  => Handler (type : Type) (eSig : Lang type) (calc : type -> Type)
        (tyAcc : Type) (m : Type -> Type) | type
  where
    handle : (env : Env m type pre)
      -> (expr : eSig tyExpr pre postK)
      -> (acc : tyAcc)
      -> (cont : (value : tyExpr)
          -> (env' : Env m type (postK value))
          -> (acc' : tyAcc)
          -> m tyRes)
      -> m tyRes

```

■ **Figure 15** Interfaces for evaluation.

```

data Tag : (type : Type) -> (value : type) -> Type where
  MkTag : RealVar type => (real : CalcRealType value) -> Tag type value

data Env : (m : Type -> Type) -> (ty : Type) -> (ctxt : Context ty calcSTy) -> Type where
  Nil : Env m type Nil
  (::) : RealVar type
    => {item : StateItem type calcSTy value}
    -> (tag : Tag type value)
    -> (rest : Env m type items)
    -> Env m type (item::items)

```

■ **Figure 16** Evaluation environment definition.

```

lookup : RealVar ty
  => (env : Env m ty ctxt)
  -> {p : (item : StateItem ty calcSTy value) -> Type}
  -> (idx : InContext value p ctxt)
  -> Tag ty value

(a) Lookup items from environment.
update : RealVar ty
  => (env : Env m ty ctxt)
  -> (idx : InContext value p ctxt)
  -> (up : (i : StateItem ty calcSTy value)
      -> p i
      -> StateItem ty calcSTy value)
  -> Env m ty (update ctxt)

drop : (env : Env m ty ctxt)
  -> (idx : InContext value p ctxt)
  -> Env m ty (drop ctxt idx)

(c) Remove items from environment.

(b) Update items from environment.

```

■ **Figure 17** Functions operating over an execution environment.

Figure 18 presents the generic function `run` that evaluates languages defined in `RESOURCES`. As arguments the function `run` takes: a closed `LangM` program (`prog`); and an initial seed for the accumulator – `init`. On successful evaluation the function returns the result of evaluating `prog` and the final state of the accumulator. The type of the function has been further constrained with `Applicative` to return the result of the evaluation within the context of the environment `m`. For pure evaluation contexts, i.e. identity, a separate `runPure` function can be defined that need not be constrained by `Applicative`.

```

run : (Applicative m, Handler type lang tyAcc m)
    => (init : tyAcc)
    -> (prog : LangM m tyExpr lang c Nil (const Nil))
    -> m (Pair tyExpr tyAcc)

```

■ **Figure 18** Run function.

4 Exemplar Uses of Resources

This section explores use of RESOURCES through the construction of three separate EDSLs. For each EDSL presented we only present salient aspects of the construction. The complete definitions are available in the accompanying artefact.

The first, **Files**, replicates the running example from Section 2 demonstrating how to build the EDSL and specify a handler for the **IO** computation context. The second, **Wireless** presents a EDSL for describing wireless connections between mobile devices, and details a **Handler** instance for constructing a *BiGraph* representation. The last EDSL, **Sessions** replicates salient aspects from, and extends the functionality, of **Sessions** an EDSL for describing communication protocols [19], and shows a simpler construction using RESOURCES.

4.1 Exemplar 1: Reasoning About Multiple File Handles

This section demonstrates how to use RESOURCES to re-implement the **FilesIOE** EDSL from Section 2.1.

4.1.1 EDSL Definition

Figure 19 presents the type-level definitions required by the EDSL. Like **FilesIOE**, there is a *single* state machine captured within the EDSL's type. The type **FH** is a singleton type acting as a meta-type for the state machine, and the type synonym **FileHandle** acts a convenient wrapper when referring to file handles. **FHStateType** is the function that calculates the state type based on **FH**, and **FileStateItem** is the type synonym for representing the EDSLs abstract states. While this construction is cumbersome for single state-machine EDSLs, Section 4.2 demonstrates how this construction can support multiple type-level state machines.

<code>data FH = MkFH</code>	<code>FHStateType : FH -> Type</code>
(a) EDSL Metatype.	<code>FHStateType _ = FileState</code>
<code>FileHandle : Type</code>	(c) Interpreter to compute state type.
<code>FileHandle = Var FH MkFH</code>	<code>FileStateItem : Type</code>
(b) Alias to represent file handles.	<code>FileStateItem = StateItem FH FHStateType MkFH</code>
	(d) Type synonym to represent state items.

■ **Figure 19** Preliminary definitions and example predicate.

Figure 20 presents the algebraic data type (**Files**) that captures the language's expressions. Notice how the definitions mirror that of **FilesIOE** from Section 2.1. Rather than use explicit case statements in anonymous functions, named functions are provided that compute the state transitions. As an example we present the function definition for **readTrans**:

```

readTrans : Either FileError String
    -> (old : Context FH FHStateType)

```

```

data Files : Lang FH FHStateType where
  Open : (fname : String)
        -> (fm : FMode)
        -> Files (Either FileError (FileHandle) old (\res => openTrans res fm old)

  Read : (hdl : FileHandle)
        -> (prf : InContext MkFH (IsOpenFor hdl R) old)
        -> Files (Either FileError String) old (\res => readTrans res old prf)

  Write : (hdl : FileHandle)
         -> (msg : String)
         -> (prf : InContext MkFH (IsOpenFor hdl W) old)
         -> Files (Maybe FileError) old (\res => writeTrans res old prf)

  Close : (hdl : FileHandle)
         -> (prf : InContext MkFH (IsHandle hdl) old)
         -> Files () old (const $ drop old prf)

  PrintLn : Show a => a -> Files () old (const old)

```

■ **Figure 20** Definition for `Files`.

```

-> InContext MkFH (IsOpenFor hdl R) old
-> Context FH FHStateType
readTrans (Right _) old _ = old
readTrans (Left _) old prf = update old prf (\i,p => closeHandle i p)

```

`Files` uses two predicates to reason about a file handle's abstract state: `IsOpenFor` and `IsHandle`. Their definition mirrors that to those provided in Figure 6. As an example, we present only the new definition for `IsOpenFor`:

```

data IsOpenFor : FileHandle -> FMode -> FileStateItem -> Type where
  FileIsOpenFor : (m : FMode)
                 -> IsOpenFor hdl m (MkStateItem MkFH hdl (Open m))

```

The language definition for `Files` is thus:

```

FILES : LANG FH FHStateType
FILES = MkLang FH FHStateType Files

```

The generic computation context `LangM` uses `LANG` instances to ensure correct embedding of EDSL expressions. Expressions can then be embedded within `LangM` using `expr` as follows:

```

openFile : (fname : String) -> (fm : FMode)
         -> LangM m (Either FileError (FileHandle)) FILES old
         (\res => openTrans res fm old)
openFile fname fm = expr $ Open fname fm

```

4.1.2 Handler for the `Files` EDSL

Figure 21 presents the handler definition for the `IO` computation context. An implementation of `RealVar` maps the singleton type `FH` to a *real* file handle. The accumulator has the unit type as this implementation of `Handler` only evaluates `File` expressions. The accumulator is not required. Each of the expression handlers realises the requisite file operations, and follows that of the `FILE` effect [10]. Within our implementation, however, our environment (`env`) keeps track of the open file handles.

```

RealVar FH where
  CalcRealType MkFH = File

Handler FH FHStateType Files () IO where
  handle env (Open fname fm) acc cont = do
    let m = case fm of {R => Read; W => WriteTruncate}
        res <- openFile fname m
        case res of
          Left err => cont (Left err) env acc
          Right fh => cont (Right MkVar) (MkTag fh::env) acc

  handle env (Read hdl prf) acc cont = do
    let MkTag fh = lookup env prf
        res <- fGetLine fh
        case res of
          Left err => cont (Left err) (update env prf (\i,p => closeHandle i p)) acc
          Right str => cont (Right str) env acc

  handle env (Write hdl str prf) acc cont = do
    let MkTag fh = lookup env prf
        res <- fPutStrLn fh str
        case res of
          Left err => cont (Just err) (update env prf (\i,p => closeHandle i p)) acc
          Right _ => cont Nothing env acc

  handle env (Close hdl prf) acc cont = do
    let MkTag fh = lookup env prf
        closeFile fh
        cont () (drop env prf) acc

  handle env (PrintLn a) acc cont = do
    printLn a
    cont () env acc

```

■ Figure 21 Handler instance for Files.

4.1.3 Example Programs

Figure 22 presents two example programs written using `Files`. The first (Figure 22a) replicates the running example presented in Figures 1 and 8. The second example (Figure 22b) demonstrates an incomplete program, indicated by the typed-hole, that will fail to type check. This is because the file has been opened for reading and we are attempting to write to the file. The typed hole is required in this example to ensure that the example can begin to type-check. `RESOURCES` ensures that the substructural checks are performed at compile time.

4.2 Exemplar 2: Constructing Domain Specific Bigraphs

This next example examines bigraphs, a mathematical model for representing the communication made between entities and said entities physical placement [40]. A bigraph comprises of a *place graph* that denotes the spatial relations between entities, and a *link graph* that denotes the communication relations. Each entity within a bigraph is typed with a domain specific construct that dictates the entity's: *arity* – number of links; and *atomicity* – containment of other entities. Existing bigraph constructions make their bigraphs abstract (entities are identifier-free) and refer to entities using singleton types [53].

Figure 23 presents a commonly used algebraic notation for bigraph specification. The standard algebraic bigraph definition embeds the link graph within an entities definition in which the type of the entity dictates the arity. The number of links K possesses is determined

```

copy : (a,b : String) -> Files m (Maybe FileError)
copy a b = do
  Right fh <- openFile a R | Left err => do {println err; pure (Just err)}
  Right s <- readString fh | Left err => do {println err; closeFile fh; pure (Just err)}
  closeFile fh
  Right fh1 <- openFile b W | Left err => do {println err; pure (Just err)}
  res <- writeString fh1 s
  case res of
    Nothing => do {closeFile fh1; pure Nothing}
    Just err => do {println err; closeFile fh1; pure (Just err)}

```

(a) Example from Figures 1 and 8.

```

copy : (a,b : String) -> Files m (Maybe FileError)
copy a b = do
  Right fh <- openFile a R | Left err => do {println err; pure (Just err)}
  Right s <- readString fh | Left err => do {println err; closeFile fh; pure (Just err)}

  writeString fh s
  ?remainder

```

(b) A failing example.

■ **Figure 22** Example instances of `Files`.

by its arity. Bigraphs can be *nested*, situated beside each other using a *merge product*, or associated together using *parallel product*. The internal structure of a bigraph entity can be abstracted away using `id`. Closure of names allows one to define internal links between entities, and free names represent external connections. Bigraphs also enjoy an expressive graphical notation which we do not detail here.

$P \cdot Q$	Nesting	(1)	<code>id</code>	Identity	(4)
$P Q$	Merge product	(2)	$K_{x,y}$	An entity of type K with names x, y	(5)
$P Q$	Parallel product	(3)	$/x P$	Closure of name x in P	(6)

■ **Figure 23** Algebraic Definition for Bigraphs.

The algebraic structure of bigraphs are general purpose and restrictions on the bigraph's shape is guided by a *system of sorts*. These sorts presents a series of side conditions on the link and place graph. Application of this system is often left as an aside from the bigraph itself. Using `RESOURCES` we can show how to build an EDSL that encapsulates the system of sorts and when interpreted produces a bigraph instance.

4.2.1 Domain Model

Existing work has introduce a bigraph model for representing Wireless Sensor Networks (WSNs) [54]. In their model they use the place graph of bigraphs to model the physical deployment of nodes, together with their configuration, and applications running on said nodes. The link graph connects data, applications, and nodes together. In this example we take a reduced version of their system of sorts to describe sending of messages between mobile devices and laptops that are connected over a wireless network. For simplicity, we restrict number of concurrent connects laptops have to ten, and mobile devices to two. Devices are located in rooms that are within buildings.

Table 1 presents the description of our example’s types and sorts. Buildings can only contain rooms and cannot be linked over for communication. Similarly, rooms can only contain devices. Devices contain only messages, and laptops and mobiles have an arity respective to their max number of connections. Messages cannot contain other entities.

■ **Table 1** Types and Sorts for representing entities in *Wireless*.

Entity	Arity	Usage Restrictions
Building	0	Complex for Rooms only.
Room	0	Complex for Devices only.
Device Laptop	10	Complex for Messages only.
Device Mobile	2	Complex for Messages only.
Messages	0	Atomic

Figure 24 presents an example bigraph instance using the system from Table 1. We situate two buildings that contain potentially many rooms next to each other, and describe some rooms within them. Within one room in the first building, a laptop is situated that is connected to another laptop in the other building, together with a mobile device (with a message) that is connected to a laptop in an adjacent room in the same building.

$$\begin{aligned} & /m/n (\text{Building} \cdot (\text{Room} \cdot (\text{Laptop}_{\{n\}} \mid (\text{Mobile}_{\{m\}} \cdot \text{Message})) \mid (\text{Room} \cdot (\text{Laptop}_{\{m\}}))) \mid \text{id}) \\ & \parallel ((\text{Building} \cdot (\text{Room} \cdot \text{Laptop}_{\{n\}} \mid \text{id}))) \end{aligned}$$

■ **Figure 24** Example Bigraph instance using algebraic notation.

4.2.2 EDSL Definition

Figure 25a presents a realisation for Table 1 using standard Idris constructs. Types are presented as an enumerated type, in which we coalesce the definition for devices. For rooms and messages we keep track of their allocation into entities, and for devices we keep track of their allocation and number of free connections. Buildings do not have an associated abstract state. The function `maxConn` calculates a devices arity, this function is used in secondary function `defState` (not defined) that constructs `Stated` instances.

```

data DTy = MOBILE | LAPTOP
data Ty = ROOM | BLDG | MSG
         | DEVICE DTy
(a) Metatypes.

maxConn : DTy -> Nat
maxConn MOBILE = 2
maxConn LAPTOP = 10
(b) Function to compute device arity.

data Stated : DTy -> Type where
  MkD : Bool -> Nat -> Stated ty
(c) State for devices.

CalcStateType : Ty -> Type
CalcStateType ROOM = Bool
CalcStateType BLDG = ()
CalcStateType (DEVICE ty) = Stated ty
CalcStateType MSG = Bool
(d) Function to compute state types.

```

■ **Figure 25** Preliminary definitions.

Figure 26 presents the language definition for *Wireless*. Introduction of entities extend the abstract state: buildings have no state; rooms and messages are initially unassigned; and devices are initialised not allocated and connection free.


```

data Wireless : Lang Ty CalcStateType where
  NewBuilding : Wireless (Var Ty BLDG) old (\lbl => MkStateItem BLDG lbl () :: old)

  NewRoom : Wireless (Var Ty ROOM) old (\lbl => MkStateItem ROOM lbl False :: old)

  NewDevice : (type : DTy)
    -> Wireless (Var Ty (DEVICE type)) old
      (\lbl => MkStateItem (DEVICE type) lbl (defState type) :: old)

  NewMessage : Wireless (Var Ty MSG) old (\lbl => MkStateItem MSG lbl False :: old)

  Insert : (varX : Var Ty x)
    -> (varY : Var Ty y)
    -> (prfValid : ValidAssign y x)
    -> (prfFree : InContext x (Unassigned x varX) old)
    -> (prfInsert : InContext y (CanAssign y varY) (update old prfFree Use))
    -> Wireless () old (const $ update (update old prfFree Use)
      prfInsert (Assign varX prfValid))

  Link : (varX : Var Ty (DEVICE typeX))
    -> (varY : Var Ty (DEVICE typeY))
    -> (prfSpaceX : InContext (DEVICE typeX) (CanConnect varX) old)
    -> (prfSpaceY : InContext (DEVICE typeY)
      (CanConnect varY)
      (update old prfSpaceX Connect))
    -> Wireless () old
      (const $ update (update old prfSpaceX Connect) prfSpaceY Connect)

  End : Wireless () old (const Nil)

```

■ **Figure 26** Definition for `Wireless`.

The constructor `Insert` is a generic expression that supports: insertion of rooms into buildings; devices into rooms; and messages into devices. For insertion of entity `varX` into `varY` to take place several checks are performed. First we check to see if the entities of type `x` and `y` are valid assertions using `ValidAssign` defined in Figure 27b. We then check to see if the child entity (`varX`) has already been inserted. The predicate `Unassigned` (Figure 28a) attests to this, and the function `Use` (Figure 28b) updates the context accordingly. The final check is to see if the parent entity (`varY`) can be assigned to. By design, the predicate `CanAssign` (Figure 27a) uses type level pattern matching to reason about abstract states that can contain other entities, and for a device that there is at least one free connection left. The function `Assign` (Figure 27c) updates the context accordingly.

```

data CanAssign : (thisValue : Ty)
  -> (thisVar : Var Ty valueThis)
  -> (item : StateItem Ty CalcStateType valueThis)
  -> Type where
  ToABuilding : CanAssign BLDG bld (MkStateItem BLDG bld ())
  ToARoom : CanAssign ROOM rm (MkStateItem ROOM rm True)
  ToADevice : CanAssign (DEVICE ty) dev (MkStateItem (DEVICE ty) dev (MkD True (S n)))

```

(a) Predicate.

```

data ValidAssign : Ty -> Ty -> Type where
  ValidBR : ValidAssign BLDG ROOM
  ValidRD : ValidAssign ROOM (DEVICE ty)
  ValidDM : ValidAssign (DEVICE ty) MSG
  Assign : Var Ty x
    -> ValidAssign v x
    -> (i : StateItem Ty CalcStateType v)
    -> CanAssign value lbl i
    -> StateItem Ty CalcStateType v

```

(b) Side-Condition.

(c) Update Function.

■ **Figure 27** Predicates and update function for reasoning about association of nodes.

```

data Unassigned : (value : Ty) -> (lbl : Var Ty value)
  -> (item : StateItem Ty CalcStateType value) -> Type where
  URoom      : Unassigned ROOM      rm (MkStateItem ROOM      rm False)
  UDevice    : Unassigned (DEVICE ty) dev (MkStateItem (DEVICE ty) dev (MkD False c))
  UMessage   : Unassigned MSG       msg (MkStateItem MSG       msg False)

```

(a) Predicate.

```

Use : (item : StateItem Ty CalcStateType value)
  -> (prf : Unassigned value lbl item)
  -> StateItem Ty CalcStateType value

```

(b) Update Function.

■ **Figure 28** Predicate and Function for Assigning Variables.

```

data CanConnect : (to : Var Ty (DEVICE type))
  -> (item : StateItem Ty CalcStateType (DEVICE type))
  -> Type where
  HasSpace : CanConnect dev (MkStateItem (DEVICE type) lbl (MkD True (S n)))

```

(a) Predicate.

```

Connect : (item : StateItem Ty CalcStateType (DEVICE type))
  -> (prf : CanConnect to item)
  -> StateItem Ty CalcStateType (DEVICE type)

```

(b) Update Function.

■ **Figure 29** Predicates and update function for reasoning about connection of devices.

Notice for `Insert` we have had to update the context twice. Once for `prfInsert`, and again in the function to calculate the new context. For each assumption we make in the type about the context we must ensure it holds for subsequent steps. Unfortunately, this can result in verbose type signatures.

Devices are linked together using `Link`. For devices to be connected we must assert, using `CanConnect` (Figure 29a), that they have free connections left to make. Like `Insert` we must also update the context for each assertion we make about each devices state. With this definition of `Link` we make no restrictions on linking devices to themselves.

4.2.3 Handler for the Bigraph EDSL

Given the algebraic notation for bigraphs their representation as an algebraic data type naturally follows. Figure 30 presents our bigraph implementation. Entities are a simple data structure capturing the arity of the entity and a unique identifier. `Entity` is parameterised by the sort type as a value. Although, we can use the arity of an entity to inform the length of a `Vect` instance to capture the link graph we must remember that bigraph’s are constructed by interpretation of an instance of `Wireless`. Interpretation must ensure that the construction of the place graph correctly matches the description from the specification, and that the entities used in the place graph are embedded correctly within the final version of the link graph. The final state of the link and place graphs will not be known until we end the specification. Therefore we must delay construction of the bigraph model until then. Thus, interpretation of `Wireless` specifications will return an intermediate bigraph representation used for constructing the algebraic bigraph representation. This is the representation presented in Figure 30.

The type, `Bigraph`, is indexed by the concrete type describing the bigraph’s “types” and specification of the place graph follows the algebraic bigraph definition. Entity arity nor the link graph are described within the `Node` constructor. Borrowing from existing algebraic graph definitions [41] links and external names are represented using `Connect` and `Outside`. `Overlay` describes the union of two bigraph descriptions into a single bigraph. Construction of a more compact algebraic bigraph model from `Bigraph` is not described here.

```

data Entity : (type : Type) -> (value : type) -> Type where
  MkEntity : (n : Nat) -> (arity : Nat) -> Entity type value

data BiGraph a = Identity | Node (Entity a value)
  | Nest (BiGraph a) (BiGraph a) | Merge (BiGraph a) (BiGraph a)
  | Par (BiGraph a) (BiGraph a)
  | Connect (BiGraph a) (BiGraph a) | Outside String
  | Overlay (BiGraph a) (BiGraph a)

```

■ **Figure 30** Naïve Algebraic Representation of a Bigraph.

Figure 31 presents the complete instance of `Handler` for `Wireless`. We use the accumulator of `Handler` to capture the bigraph instance being constructed, and a counter to generate fresh identifiers. An instance of `RealVar` translates variables into `Entity` instances, the type `Ty` has been reused for entity types. Variables are turned into entities and extend the environment for each new variable definition. Insertion of entities creates a `Nest` instruction and each variable definition is inserted into a `Node` constructor. Although the definition of the handler for `Insert` looks repetitive, dependent pattern matching on the side-condition (`prfValid`) is required to ensure that the correct proofs are considered at the type level [38]. Each lookup and proof used for each case have different types. Further, for each operation that updates the context we must also update the environment accordingly. With the updates to the environment mirroring the updates made at the type level. The `Overlay` instruction combines the existing bigraph (`acc`) with a new nesting of parent and child. Interpretation of `Link` follows that for `Insert` by updating the environment for each change in the context, and appends to the accumulated bigraph using `Overlay`, a new edge in the link graph using `Connect`.

4.2.4 Example Bigraph Instances

Figure 32 illustrates how several example bigraphs can be specified using `Wireless`. The type-synonym `WirelessDesc` sets the expected initial and end states – cf. `Files` in Section 4.1. Figure 32a replicates the example from Figure 24. Figure 32b presents a failing example that will not type-check as the domain model specifies that models only support two connections. The final `link` expression will fail to type-check as the abstract state for `mobileA` will have decremented the number of free connections to zero.

4.3 Exemplar 3: Global Session Descriptions

Multi-Party Session Types (MPST) are a typing discipline that allows formal protocol narrations to dictate the type checking process such that implementations of the protocol are *known* to adhere to a given formal narration [30]. Global session types present an overview of the interactions made between entities, and entities have local types that describes their known interactions. Existing work has seen to extend MPST implementation and theory to support reasoning on message values [30, 61, 6]. When looking to realise global session types in a dependently typed language care must be taken that values introduced in the description are used by roles that know about the value.

`SESSIONS` is an EDSL for describing global session descriptions [19]. Figure 33 illustrates how `SESSIONS` can be written using `RESOURCES` as `Sessions`, and Figure 34 details accompanying data types and functions. For brevity, we have not included the creation of value dependent messages. We have, however, extended the EDSL with expressions to reason explicitly about channels, that borrows from existing work [31]. The type of `Sessions` is

```

RealVar Ty where
  CalcRealType ROOM      = Entity Ty ROOM
  CalcRealType BLDG      = Entity Ty BLDG
  CalcRealType (DEVICE x) = Entity Ty (DEVICE x)
  CalcRealType MSG       = Entity Ty MSG

Handler Ty CalcStateType Wireless (Nat, BiGraph Ty) (Basics.id) where
  handle env NewBuilding (ctr,g) cont =
    cont MkVar (MkTag (MkEntity ctr Z)::env) (S ctr,g)
  handle env NewRoom (ctr,g) cont = cont MkVar (MkTag (MkEntity ctr Z)::env) (S ctr,g)
  handle env (NewDevice ty) (ctr,g) cont =
    cont MkVar (MkTag (MkEntity ctr (maxConn ty))::env) (S ctr,g)
  handle env NewMessage (ctr,g) cont = cont MkVar (MkTag (MkEntity ctr Z)::env) (S ctr,g)

  handle env (Insert varX varY prfValid prfFree prfInsert) (ctr,g) cont with (prfValid)
    handle env (Insert varX varY prfValid prfFree prfInsert) (ctr,g) cont | ValidBR = do
      let MkTag rm = lookup env prfFree
          let env' = (update env prfFree Use)
          let MkTag bld = lookup env' prfInsert
          let env'' = update env' prfInsert (Assign varX ValidBR)
          cont () env'' (ctr,Overlay (Nest (Node bld) (Node rm)) g)
    handle env (Insert varX varY prfValid prfFree prfInsert) (ctr,g) cont | ValidRD = do
      let MkTag dev = lookup env prfFree
          let env' = (update env prfFree Use)
          let MkTag rm = lookup env' prfInsert
          let env'' = update env' prfInsert (Assign varX ValidRD)
          cont () env'' (ctr,Overlay (Nest (Node rm) (Node dev)) g)
    handle env (Insert varX varY prfValid prfFree prfInsert) (ctr,g) cont | ValidDM = do
      let MkTag msg = lookup env prfFree
          let env' = (update env prfFree Use)
          let MkTag dev = lookup env' prfInsert
          let env'' = update env' prfInsert (Assign varX ValidDM)
          cont () env'' (ctr,Overlay (Nest (Node dev) (Node msg)) g)

  handle env (Link varX varY prfSpaceX prfSpaceY) (ctr,g) cont = do
    let MkTag x = lookup env prfSpaceX
        let env' = update env prfSpaceX Connect
        let MkTag y = lookup env' prfSpaceY
        let env'' = update env' prfSpaceY Connect
        cont () env'' (ctr,Overlay (Connect (Node x) (Node y)) g)

  handle env End (ctr,g) cont = cont () Nil (ctr,g)

```

■ **Figure 31** Handler instance for `Wireless`.

further parameterised by a list of participants in the protocol, allowing the EDSL to utilise this information for each expression. `Sessions` expressions include message creation, channel construction and destruction, sending of messages, allowing access to message values, and termination of session descriptions.

Central to the operation of `Sessions` is reasoning about the abstract state associated with messages and communication channels. Messages have metatype `DATA` capturing the type of the message, and an associated state listing the actors aware of the message. Channels have metatype `CHAN` capturing the involved actors, and an associated state denoting the connection state: `Bound` or `Free`. The function `CalcStateType` maps meta types to concrete state types.

The construction of `Sessions` follows that of `SESSIONS` but in a more general framework. Message creation extends the list of state with a new abstract state asserting that the creator `a` knows of the message, and the expression's use is restricted to actors listed in the descriptions type. Similarly channel creation extends the list of states with a new abstract state asserting the channel is `Bound`, and restricts channel creation to actors listed in the description's type.

```

example : WirelessDesc m
example = do
  buildingA <- newBuilding
  buildingB <- newBuilding

  roomA <- newRoom
  roomB <- newRoom
  roomC <- newRoom

  laptopA <- newDevice LAPTOP
  laptopB <- newDevice LAPTOP
  laptopC <- newDevice LAPTOP
  mobile <- newDevice MOBILE

  msg <- newMessage

  insert roomA buildingA
  insert laptopA roomA
  insert mobile roomA
  insert msg mobile
  insert roomB buildingA
  insert laptopB roomB
  insert roomC buildingB
  insert laptopC roomC

  link laptopA laptopC
  link mobile laptopB

end

```

```

example : WirelessDesc m
example = do
  buildingA <- newBuilding
  roomA <- newRoom

  insert roomA buildingA

  mobileA <- newDevice MOBILE
  mobileB <- newDevice MOBILE
  mobileC <- newDevice MOBILE
  mobileD <- newDevice MOBILE

  insert mobileA roomA
  insert mobileB roomA
  insert mobileC roomA
  insert mobileD roomA

  link mobileA mobileB
  link mobileA mobileC
  link mobileA mobileD
end

```

(a) Example Bigraph from Figure 24.

(b) Example Failing Bigraph.

■ **Figure 32** Example Specifications using `Wireless`.

This specification implies that we are free to make connections between any two actors in `ps`. We could add a predicate to `Sessions` that restricts channel creation to specific pairings of actors. Closing a channel changes the channel’s abstract state to `Free`. Sending messages along a channel requires an active channel guaranteed by the predicate `ChannelHasState`, and proof (using `KnowsData`) that the sender (`s`) knows about the message. Once a message has been sent the abstract state of the message is updated to reflect that the receiver is now aware of the message.

Figure 35 presents the definition for these and other predicates used in `Sessions`. The expression `ReadMsg` facilitates reasoning using message values that are known to all participants. Session descriptions conclude if all abstract states are in a valid end state. For messages, this is immaterial and for connections they must have been closed.

It is reasonable to assume that we can define a projection function as an instance of `Handler`. The type for `handle`, however, requires that we build a continuation that can be applied to a value associated with the expression. When projecting global types in a multi-party session some expressions are irrelevant if the role being projected for is not involved [14]. The type for `handle` is too constrained for `Sessions` implementation. Future work will be to investigate how a projection function for `Sessions` can be constructed.

Figure 36 present several example session descriptions. The function `Session` is a type-synonym to restrict the starting and ending type-level context to `Nil`. Figure 36a models the salient aspects of the TCP handshake [51]. Here Alice and Bob establish a channel, and Alice sends to Bob a sequence number (`x`) that Bob must return incremented by one. Similarly, Bob sends Alice a sequence (`y`) that Alice must return incremented by one. In our description we use dependent pairs to reason about the message contents.

20:22 A Framework for Resource Dependent EDSLs in a Dependently Typed Language

```

data Sessions : (participants : List Actor) -> Lang Ty CalcStateType where

NewData : (a : Actor) -> (type : Type) -> (prf : Elem a ps)
  -> Sessions ps (Var Ty (DATA type)) old
  (\lbl => MkStateItem (DATA type) lbl (MkDataState [a] type) :: old)

NewConnection : (a,b : Actor) -> (prfS : Elem a ps) -> (prfR : Elem b ps)
  -> Sessions ps (Var Ty (CHAN (a,b))) old
  (\lbl => MkStateItem (CHAN (a,b)) lbl (MkChanState Bound) :: old)

EndConnection : (chan : Var Ty (CHAN (a,b)))
  -> (prf : InContext (CHAN (a,b)) (ChannelHasState Bound chan) old)
  -> Sessions ps () old (const $ update old prf (SetChannelState Free))

SendLeft : (chan : Var Ty (CHAN (s,r)))
  -> (msg : Var Ty (DATA type))
  -> (prfActive : InContext (CHAN (s,r)) (ChannelHasState Bound chan) old)
  -> (prfKnows : InContext (DATA type) (KnowsData s msg) old)
  -> Sessions ps () old (const $ update old prfKnows (ExpandWhoKnows r))

SendRight : (chan : Var Ty (CHAN (s,r)))
  -> (msg : Var Ty (DATA type))
  -> (prfActive : InContext (CHAN (s,r)) (ChannelHasState Bound chan) old)
  -> (prfKnows : InContext (DATA type) (KnowsData r msg) old)
  -> Sessions ps () old (const $ update old prfKnows (ExpandWhoKnows s))

ReadMsg : (msg : Var Ty (DATA type))
  -> (prf : InContext (DATA type) (AllKnow ps msg) old)
  -> Sessions ps type old (const old)

StopSession : AllContext EndState old -> Sessions ps () old (const Nil)

```

■ **Figure 33** An EDSL for describing Global Multi-Party Session Types.

<pre> data Actor = MkActor String data Usage = Free Bound </pre> <p>(a) Actors, Usage, and Metatypes.</p>	<pre> data ChanState = MkChanState Usage data DataState = MkDataState (List Actor) Type </pre> <p>(b) Abstract States.</p>
<pre> data Ty = CHAN (Actor, Actor) DATA Type </pre> <p>(c) Function to compute state types.</p>	<pre> CalcStateType : Ty -> Type CalcStateType (CHAN _) = ChanState CalcStateType (DATA _) = DataState </pre>

■ **Figure 34** Core accompanying data types and functions.

Figures 36b and 36c present two examples that fail to type-check. The first Figure 36b demonstrates how sending on the wrong channel will result in a type error. Here Alice is not involved in the communication between Bob and Charlie. The second example Figure 36c shows an example that will fail as the message (`m`) is not yet known by all participants.

5 Related Work

The implementation of RESOURCES builds upon existing techniques developed for Effects [11] that realise well studied theoretical models [1, 43, 50]. These models were realised in a dependently typed language using straightforward idiomatic constructs: Hoare monads as a parameterised data type; and algebraic effect handlers using interfaces.

```

data ChannelHasState : (assumedState : Usage)
  -> (chan : Var Ty (CHAN (s,r)))
  -> (actual : StateItem Ty CalcStateType (CHAN (s,r)))
  -> Type where
  ChanHasState : ChannelHasState st ch (MkStateItem (CHAN (s,r)) ch (MkChanState st))

```

(a) Asserting Channel State.

```

data AllKnow : (as : List Actor)
  -> (var : Var Ty (DATA type))
  -> (item : StateItem Ty CalcStateType (DATA ty))
  -> Type where
  NilKnows : (prf : Elem x as)
    -> AllKnow [x] msg (MkStateItem (DATA ty) msg (MkDataState as ty))
  ConsKnows : (prf : Elem x as)
    -> (later : AllKnow xs msg (MkStateItem (DATA type) msg (MkDataState as ty)))
    -> AllKnow (x::xs) msg (MkStateItem (DATA type) msg (MkDataState as ty))

```

(b) Asserting that all participants know a value.

```

data KnowsData : (actor : Actor)
  -> (var : Var Ty (DATA type))
  -> (item : StateItem Ty CalcStateType (DATA type))
  -> Type where
  DoesKnow : (prf : x = y) -> (prfE : Elem x actors)
    -> KnowsData y var (MkStateItem (DATA type) var (MkDataState actors type))

```

(c) Asserting that a participant know a value.

```

data EndState : (ty : Ty) -> StateItem Ty CalcStateType ty -> Type where
  EndData : EndState (DATA type) state
  EndConn : EndState (CHAN (s,r)) (MkStateItem (CHAN (s,r)) lbl (MkChanState Free))

```

(d) Asserting final end states.

■ **Figure 35** Predicates used in `Sessions`.

5.1 Theoretical-Oriented Approaches

First we examine other theoretical approaches to realising substructural type-systems for EDSLs that use expressive logics as a base formalism.

Hoare Type Theory. *Hoare Type Theory* [43] has been used to describe programs with substructural type systems [7]. Here types are associated with Hoare triples that are translated to refinement types [23, 27] to ensure triple satisfaction. Ynot is an extension of the Coq proof assistant to provide reasoning about programs using Hoare Type Theory [42]. Similar work has presented a variant of the State Monad that provides Hoare style reasoning on the captured state [60]. Our approach also utilises Hoare triples but not to reason about individual types *per se*, but rather about the entire type-level state of our program. This is much similar to existing work [8, 7] in which the authors were restricted to reasoning about the program’s state, described as a state monad, in its entirety. Use of quantifiers over our abstract state allows us to reason about specific aspects of a program’s state.

Typestates. *Typestates* [56, 22, 5] have been shown to provide a formal basis for building substructural type-systems [39]. Using their approach the authors also show how to incorporate behavioural typing [16] as well. Here each type in their formalism is associated with a type-level state and value level operations apply, at the type level, state transitions to the modelled state. This is a more formal treatment compared to our approach, however, we acknowledge the similarity in associating types with a type-level state. Rather than use typestates as a base formalism we utilise parameterised monads.

```

TCPHandshake : Session m [Alice, Bob]
TCPHandshake = do
  chan <- setup Alice Bob

  m1 <- msg Alice (Packet, Nat)
  sendLeft chan m1

  (p,x) <- read m1
  m2 <- msg Bob
    (Packet, (x' ** x' = S x),
             Nat)

  sendRight chan m2

  (p,xplus,y) <- read m2

  m3 <- msg Alice
    (Packet, (x' ** x' = S x),
             (y' ** y' = S y))

  sendLeft chan m3
  destroy chan
end
(a) TCP Handshake.

WrongChan : Session m [Alice,Bob,Charlie]
WrongChan = do
  chan <- setup Alice Bob
  net <- setup Bob Charlie

  m <- msg Alice String
  sendRight net m
  ?end
(b) Sending on a wrong channel.

UnableToRead : Session m
               [Alice,Bob,Charlie]
UnableToRead = do
  chan <- setup Alice Bob

  m <- msg Alice String
  sendLeft chan m

  val <- read m
  ?end
(c) Invalid Read Access.

```

■ **Figure 36** Example Global Session Descriptions.

Separation Logics. *Separation logic* [45] has been used to provide another formal treatment towards customising standard substructural type-systems with custom resources [34]. This work supports customisation of resources, and controls on said resource to be specified on a *per-module* or *per-library* basis. Similarly, the authors use state-transition systems by way of commutative monoids, to reason about substructural properties.

Very recent work has investigated the use of separation logics to build intrinsically-typed definitional interpreters for linear/session-typed languages [52]. Developed in Agda [44] the authors present a collection of reusable and composable abstractions to support interpreter construction. Our approach differs in that we ground our work in hoare logic and provide a singular unified framework to capture common language expressions common to all EDSLs, and integrated support for reasoning on abstract program state. We will carefully study the use of separation logics and see how RESOURCES can be bettered. Of interest will be the ability of the author’s approach to realise global MPST.

Quantitative Type-Systems. Substructural Type-Systems [65] support various different styles of reasoning about variable usage at the type level. Linear typing providing *exactly once* semantics [64, 63], and Affine systems *at most once* [62, 15]. Generally speaking, *Quantitative Type-Theory* (QTT) [2] provides a more general framework to reason about resource usage. However it is not clear how state-based substructural properties (cf. nesting for Bigraphs – Section 4.2.1) can be modelled within QTT. Their substructural properties are not all about quantitative usage. Regardless, QTT is a promising direction for reasoning about quantitative resource usage.

5.2 Practical-Oriented Approaches

RESOURCES is highly dependent on Idris specific features such as interfaces and proof search, as well as dependent types. Realising RESOURCES in dissimilar languages would require more complicated work arounds to realise a similar framework, or require direct modification to the compiler. Like Effects, RESOURCES is a *plain-old-library* that does not require any

language extensions, and leverages a language in which dependent types were not retrofitted. This gives us static compile time guarantees that our framework, and thus our EDSLs, are well-typed. We now examine other practical approaches that involve expressive host languages which support construction of bespoke substructural type-systems within the host language itself.

Substructural Type-Systems. There exist several general purpose languages that provide full or experimental support for substructural type-systems. Linear typing has been realised for ATS [55]. Clean has implemented uniqueness typing [21] that influenced Rust’s ownership types [35, 36]. An extension is being developed for Haskell that leverages linearity for correct variable usage [4]. Interestingly, this extension also uses a parameterised type to replicate typestates when reasoning about a socket example. Idris itself has experimental support for linear typing [37], and Clean’s Uniqueness types [14]. Future iterations of Idris³, however, will support Quantitative Type-Theory [2]. As we described in Section 5.1, it is not clear how we can use these quantitative systems to describe state-based substructural properties.

Expressive Type-Systems. Construction of domain specific substructural type systems for EDSLs can be achieved using other as expressive non-dependently typed host languages. Racket is a general purpose language that supports EDSL creation through fine-grained control over the language’s type-system [25]. The original version of F^* was a general purpose language with value-dependent types [57, 58]. Whereas Idris provides full-spectrum dependent types, F^* provides value-dependencies using refinement types. F^* was extended to provide better support for dependently typed and effectful programming [59]. Such languages provide novel, alternate, environment in which to construct “value-dependently-typed” programs. How the approach behind RESOURCES is transferable to these languages is worth investigating.

Dependent-Type Systems. Although, the framework has been realised using Idris the techniques presented are agnostic to dependently typed languages. Any other dependently typed language that supports full-spectrum dependent types, such as Agda [44], will be suitable for implementing the ideas. It has been shown how Dependent Haskell [66] can realise Idris’ Effects library [24, § 3.2.3]. Existing work has investigated embedding linear type-systems for EDSLs into Haskell [49]. In their implementation the author’s make extensive use of Haskell’s typeclass mechanism, a *Higher Order Abstract Syntax* embedding, and Dependent Haskell [48].

ST is an improvement upon Effects by not only associating resources with variables, but facilitating vertical and horizontal effect composition [12]. ST is a resource dependent EDSL, and makes extensive use of Idris’ Interface mechanism for effect definition. RESOURCES sits in between these two implementations, borrowing the algebraic language definition from Effects and associating abstract state with variables from ST. We position RESOURCES as a framework for defining EDSLs with domain specific substructural type-systems. ST and Effects are general purpose.

³ <https://github.com/edwinb/idris2>

6 Future Work

RESOURCES is a promising framework for constructing EDSLs with interesting type-systems. However, there are a few limitations to our current approach.

Each EDSL requires that the complete set of resource types to be used be known at EDSL design time. Our EDSL are closed worlds. This impacts upon the composability of resources between EDSL instances. For example, state resources have to be created per EDSL. This limitations originate from the framework's design. It was not designed for resource reuse in mind. A promising direction will be to look at how Idris' interfaces can look to better the specification and use of resources in EDSLs.

Similarly, our type-level state holds too much information. If we were to call out to sub-programs we must be careful about the effect that the resulting state (of the callee program) has on the caller program. We see this in the design of `Files` (Section 4.1) when passing around file handles we need to ensure that closed files remain closed. While one can state that sub-programs are closed it will be interesting to investigate how to deal with interactions of states between programs. When looking at program composition, and resource reuse between EDSLs, how we interact with type-level state is important. Our use of Hoare logics prohibits the inspection, individually, of resources in isolation. By basing the framework on separation logics rather than Hoare triples, we can look to address these limitations.

Updating the type-level context multiple times in a type signature, can lead to a more verbose style of type-level programming. For example, consider the type-signatures for `Link` and `Insert` for `Wireless` in Section 4.2.2. The limitation here is Idris' own syntax: type signatures are not equivalent to a function body. Future work will be to see how we can reason better about the transformations made to the abstract state within a type signature.

7 Conclusions

RESOURCES has been developed to explore construction of EDSLs with substructural type-systems supporting autonomic management of domain specific abstract resources and type-level reasoning on such resources. Idris' support for *auto-implicit* arguments allows languages to be presented cleanly, where proofs that properties hold are hidden but present during type-checking. Resources and their state need not be listed explicitly at the type-level.

We have demonstrated the use of RESOURCES through construction of several exemplar EDSLs. Type-level predicates provided compile time guarantees over various substructural properties. Providing static compile time checks that correct EDSL instances are constructed. We have demonstrated how `Handler` instances can: run interactive programs – Section 4.1.2; and construct data types – Section 4.2.3. When we construct data structures, however, the correctness-by-construction guarantees are not necessarily carried over. It will be interesting to see how we can use RESOURCES to do so. This would be useful for our Bigraph example. This is future work. Further, we have seen when the `Handler` interface was not enough for our needs (Section 4.3), and noted limitations on program and resource composition – Section 6.

We are using RESOURCES to develop EDSLs for reasoning about the structural and behavioural aspects of System-on-a-Chip Designs. Within these languages a substructural type-system allows one to constrain expressions using type-level resources derived from finite sources and behavioural specifications. Ensuring, for example, that ports can be connected to only once, and that interfaces and connections are well-formed respective to a given specification. RESOURCES helps by providing a common framework to explore different model designs without specifying the same boilerplate again and again.

References

- 1 Robert Atkey. Parameterised notions of computation. *J. Funct. Program.*, 19(3-4):335–376, 2009. doi:10.1017/S095679680900728X.
- 2 Robert Atkey. Syntax and semantics of quantitative type theory. In Anuj Dawar and Erich Grädel, editors, *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2018, Oxford, UK, July 09-12, 2018*, pages 56–65. ACM, 2018. doi:10.1145/3209108.3209189.
- 3 Lennart Augustsson and Magnus Carlsson. An Exercise in Dependent Types: A Well-Typed Interpreter. In *In Workshop on Dependent Types in Programming, Gothenburg*, 1999.
- 4 Jean-Philippe Bernardy, Mathieu Boespflug, Ryan R. Newton, Simon Peyton Jones, and Arnaud Spiwack. Linear haskell: practical linearity in a higher-order polymorphic language. *PACMPL*, 2(POPL):5:1–5:29, 2018. doi:10.1145/3158093.
- 5 Kevin Bierhoff and Jonathan Aldrich. Modular typestate checking of aliased objects. In Richard P. Gabriel, David F. Bacon, Cristina Videira Lopes, and Guy L. Steele Jr., editors, *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2007, October 21-25, 2007, Montreal, Quebec, Canada*, pages 301–320. ACM, 2007. doi:10.1145/1297027.1297050.
- 6 Laura Bocchi, Kohei Honda, Emilio Tuosto, and Nobuko Yoshida. A theory of design-by-contract for distributed multiparty interactions. In *CONCUR 2010 - Concurrency Theory, 21th International Conference, CONCUR 2010, Paris, France, August 31-September 3, 2010. Proceedings*, pages 162–176, 2010. doi:10.1007/978-3-642-15375-4_12.
- 7 Johannes Borgström, Juan Chen, and Nikhil Swamy. Verifying stateful programs with substructural state and hoare types. In Ranjit Jhala and Wouter Swierstra, editors, *Proceedings of the 5th ACM Workshop Programming Languages meets Program Verification, PLPV 2011, Austin, TX, USA, January 29, 2011*, pages 15–26. ACM, 2011. doi:10.1145/1929529.1929532.
- 8 Johannes Borgström, Andrew D. Gordon, and Riccardo Pucella. Roles, stacks, histories: A triple for hoare. *J. Funct. Program.*, 21(2):159–207, 2011. doi:10.1017/S0956796810000134.
- 9 Edwin Brady. Idris, a general-purpose dependently typed programming language: Design and implementation. *J. Funct. Program.*, 23(5):552–593, 2013. doi:10.1017/S095679681300018X.
- 10 Edwin Brady. Programming and reasoning with algebraic effects and dependent types. In Greg Morrisett and Tarmo Uustalu, editors, *ACM SIGPLAN International Conference on Functional Programming, ICFP’13, Boston, MA, USA - September 25 - 27, 2013*, pages 133–144. ACM, 2013. doi:10.1145/2500365.2500581.
- 11 Edwin Brady. Resource-dependent algebraic effects. In Jurriaan Hage and Jay McCarthy, editors, *Trends in Functional Programming - 15th International Symposium, TFP 2014, Soesterberg, The Netherlands, May 26-28, 2014. Revised Selected Papers*, volume 8843 of *Lecture Notes in Computer Science*, pages 18–33. Springer, 2014. doi:10.1007/978-3-319-14675-1_2.
- 12 Edwin Brady. State Machines All The Way Down: An Architecture for Dependently Typed Applications. Unpublished Draft., 2016.
- 13 Edwin Brady. *Type-Driven Development with Idris*. Manning, 1st edition, 2016.
- 14 Edwin Brady. Type-driven development of concurrent communicating systems. *Computer Science (AGH)*, 18(3), 2017. doi:10.7494/csci.2017.18.3.1413.
- 15 Michele Bugliesi, Stefano Calzavara, Fabienne Eigner, and Matteo Maffei. Affine refinement types for secure distributed programming. *ACM Trans. Program. Lang. Syst.*, 37(4):11:1–11:66, 2015. doi:10.1145/2743018.
- 16 Luís Caires and João Costa Seco. The type discipline of behavioral separation. In Roberto Giacobazzi and Radhia Cousot, editors, *The 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL ’13, Rome, Italy - January 23 - 25, 2013*, pages 275–286. ACM, 2013. doi:10.1145/2429069.2429103.
- 17 Elias Castegren and Tobias Wrigstad. Reference capabilities for concurrency control. In Shriram Krishnamurthi and Benjamin S. Lerner, editors, *30th European Conference on Object-Oriented Programming, ECOOP 2016, July 18-22, 2016, Rome, Italy*, volume 56 of *LIPICs*, pages 5:1–5:26. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2016. doi:10.4230/LIPICs.ECOOP.2016.5.

- 18 N.G de Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the church-rosser theorem. *Indagationes Mathematicae (Proceedings)*, 75(5):381–392, 1972.
- 19 Jan de Muijnck-Hughes, Edwin Brady, and Wim Vanderbauwhede. Value-dependent session design in a dependently typed language. In Francisco Martins and Dominic Orchard, editors, *Proceedings Programming Language Approaches to Concurrency- and Communication-cEntric Software, PLACES@ETAPS 2019, Prague, Czech Republic, 7th April 2019*, volume 291 of *EPTCS*, pages 47–59, 2019. doi:10.4204/EPTCS.291.5.
- 20 Jan de Muijnck-Hughes and Wim Vanderbauwhede. A typing discipline for hardware interfaces. In Alastair F. Donaldson, editor, *33rd European Conference on Object-Oriented Programming, ECOOP 2019, July 15-19, 2019, London, United Kingdom*, volume 134 of *LIPICs*, pages 6:1–6:27. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019. doi:10.4230/LIPICs.ECOOP.2019.6.
- 21 Edsko de Vries, Rinus Plasmeijer, and David M. Abrahamson. Uniqueness typing simplified. In Olaf Chitil, Zoltán Horváth, and Viktória Zsók, editors, *Implementation and Application of Functional Languages, 19th International Workshop, IFL 2007, Freiburg, Germany, September 27-29, 2007. Revised Selected Papers*, volume 5083 of *Lecture Notes in Computer Science*, pages 201–218. Springer, 2007. doi:10.1007/978-3-540-85373-2_12.
- 22 Robert DeLine and Manuel Fähndrich. Typestates for objects. In Martin Odersky, editor, *ECOOP 2004 - Object-Oriented Programming, 18th European Conference, Oslo, Norway, June 14-18, 2004, Proceedings*, volume 3086 of *Lecture Notes in Computer Science*, pages 465–490. Springer, 2004. doi:10.1007/978-3-540-24851-4_21.
- 23 Ewen Denney. Refinement types for specification. In David Gries and Willem P. de Roever, editors, *Programming Concepts and Methods, IFIP TC2/WG2.2,2.3 International Conference on Programming Concepts and Methods (PROCOMET '98) 8-12 June 1998, Shelter Island, New York, USA*, volume 125 of *IFIP Conference Proceedings*, pages 148–166. Chapman & Hall, 1998.
- 24 Richard A. Eisenberg. Dependent types in haskell: Theory and practice. *CoRR*, abs/1610.07978, 2016. arXiv:1610.07978.
- 25 Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, Shriram Krishnamurthi, Eli Barzilay, Jay A. McCarthy, and Sam Tobin-Hochstadt. A programmable programming language. *Commun. ACM*, 61(3):62–71, 2018. doi:10.1145/3127323.
- 26 Martin Fowler. *Domain-Specific Languages*. Addison-Wesley Signature Series. Addison-Wesley Professional, 1 edition, October 2010.
- 27 Timothy S. Freeman and Frank Pfenning. Refinement types for ML. In David S. Wise, editor, *Proceedings of the ACM SIGPLAN'91 Conference on Programming Language Design and Implementation (PLDI), Toronto, Ontario, Canada, June 26-28, 1991*, pages 268–277. ACM, 1991. doi:10.1145/113445.113468.
- 28 Colin S. Gordon, Matthew J. Parkinson, Jared Parsons, Aleks Bromfield, and Joe Duffy. Uniqueness and reference immutability for safe parallelism. In Gary T. Leavens and Matthew B. Dwyer, editors, *Proceedings of the 27th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2012, part of SPLASH 2012, Tucson, AZ, USA, October 21-25, 2012*, pages 21–40. ACM, 2012. doi:10.1145/2384616.2384619.
- 29 C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, 1969. doi:10.1145/363235.363259.
- 30 Kohei Honda, Nobuko Yoshida, and Marco Carbone. Multiparty asynchronous session types. *J. ACM*, 63(1):9:1–9:67, 2016. doi:10.1145/2827695.
- 31 Raymond Hu and Nobuko Yoshida. Explicit connection actions in multiparty session types. In *Fundamental Approaches to Software Engineering - 20th International Conference, FASE 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings*, pages 116–133, 2017. doi:10.1007/978-3-662-54494-5_7.

- 32 Wei Huang, Werner Dietl, Ana Milanova, and Michael D. Ernst. Inference and checking of object ownership. In James Noble, editor, *ECOOP 2012 – Object-Oriented Programming – 26th European Conference, Beijing, China, June 11-16, 2012. Proceedings*, volume 7313 of *Lecture Notes in Computer Science*, pages 181–206. Springer, 2012. doi:10.1007/978-3-642-31057-7_9.
- 33 Ohad Kammar, Sam Lindley, and Nicolas Oury. Handlers in action. In Greg Morrisett and Tarmo Uustalu, editors, *ACM SIGPLAN International Conference on Functional Programming, ICFP’13, Boston, MA, USA - September 25 - 27, 2013*, pages 145–158. ACM, 2013. doi:10.1145/2500365.2500590.
- 34 Neelakantan R. Krishnaswami, Aaron Turon, Derek Dreyer, and Deepak Garg. Superficially substructural types. In Peter Thiemann and Robby Bruce Findler, editors, *ACM SIGPLAN International Conference on Functional Programming, ICFP’12, Copenhagen, Denmark, September 9-15, 2012*, pages 41–54. ACM, 2012. doi:10.1145/2364527.2364536.
- 35 Amit A. Levy, Michael P. Andersen, Bradford Campbell, David E. Culler, Prabal Dutta, Branden Ghena, Philip Levis, and Pat Pannuto. Ownership is theft: experiences building an embedded OS in rust. In Shan Lu, editor, *Proceedings of the 8th Workshop on Programming Languages and Operating Systems, PLOS 2015, Monterey, California, USA, October 4, 2015*, pages 21–26. ACM, 2015. doi:10.1145/2818302.2818306.
- 36 Nicholas D. Matsakis and Felix S. Klock II. The rust language. In Michael Feldman and S. Tucker Taft, editors, *Proceedings of the 2014 ACM SIGAda annual conference on High integrity language technology, HILT 2014, Portland, Oregon, USA, October 18-21, 2014*, pages 103–104. ACM, 2014. doi:10.1145/2663171.2663188.
- 37 Conor McBride. I got plenty o’ nuttin’. In Sam Lindley, Conor McBride, Philip W. Trinder, and Donald Sannella, editors, *A List of Successes That Can Change the World - Essays Dedicated to Philip Wadler on the Occasion of His 60th Birthday*, volume 9600 of *Lecture Notes in Computer Science*, pages 207–233. Springer, 2016. doi:10.1007/978-3-319-30936-1_12.
- 38 Conor McBride and James McKinna. The view from the left. *J. Funct. Program.*, 14(1):69–111, 2004. doi:10.1017/S0956796803004829.
- 39 Filipe Militão, Jonathan Aldrich, and Luís Caires. Substructural typestates. In Nils Anders Danielsson and Bart Jacobs, editors, *Proceedings of the 2014 ACM SIGPLAN Workshop on Programming Languages meets Program Verification, PLPV 2014, January 21, 2014, San Diego, California, USA, Co-located with POPL ’14*, pages 15–26. ACM, 2014. doi:10.1145/2541568.2541574.
- 40 Robin Milner. *The Space and Motion of Communicating Agents*. Cambridge University Press, 2009.
- 41 Andrey Mokhov. Algebraic graphs with class (functional pearl). In *Proceedings of the 10th ACM SIGPLAN International Symposium on Haskell, Oxford, United Kingdom, September 7-8, 2017*, pages 2–13, 2017. doi:10.1145/3122955.3122956.
- 42 Aleksandar Nanevski, Greg Morrisett, Avraham Shinnar, Paul Govereau, and Lars Birkedal. Ynot: dependent types for imperative programs. In *Proceeding of the 13th ACM SIGPLAN international conference on Functional programming, ICFP 2008, Victoria, BC, Canada, September 20-28, 2008*, pages 229–240, 2008. doi:10.1145/1411204.1411237.
- 43 Aleksandar Nanevski, J. Gregory Morrisett, and Lars Birkedal. Hoare type theory, polymorphism and separation. *J. Funct. Program.*, 18(5-6):865–911, 2008. doi:10.1017/S0956796808006953.
- 44 Ulf Norell. Dependently typed programming in agda. In Andrew Kennedy and Amal Ahmed, editors, *Proceedings of TLDI’09: 2009 ACM SIGPLAN International Workshop on Types in Languages Design and Implementation, Savannah, GA, USA, January 24, 2009*, pages 1–2. ACM, 2009. doi:10.1145/1481861.1481862.
- 45 Peter W. O’Hearn. Separation logic. *Commun. ACM*, 62(2):86–95, 2019. doi:10.1145/3211968.

- 46 Dominic A. Orchard and Tomas Petricek. Embedding effect systems in haskell. In Wouter Swierstra, editor, *Proceedings of the 2014 ACM SIGPLAN symposium on Haskell, Gothenburg, Sweden, September 4-5, 2014*, pages 13–24. ACM, 2014. doi:10.1145/2633357.2633368.
- 47 Johan Östlund and Tobias Wrigstad. Multiple aggregate entry points for ownership types. In James Noble, editor, *ECOOP 2012 - Object-Oriented Programming - 26th European Conference, Beijing, China, June 11-16, 2012. Proceedings*, volume 7313 of *Lecture Notes in Computer Science*, pages 156–180. Springer, 2012. doi:10.1007/978-3-642-31057-7_8.
- 48 Jennifer Paykin. *Linear/non-Linear Types for Embedded Domain-Specific Languages*. PhD thesis, University of Pennsylvania, 2018. Publicly Accessible Penn Dissertations. 2752. URL: <https://repository.upenn.edu/edissertations/2752>.
- 49 Jennifer Paykin and Steve Zdancewic. The linearity monad. In *Proceedings of the 10th ACM SIGPLAN International Symposium on Haskell, Oxford, United Kingdom, September 7-8, 2017*, pages 117–132, 2017. doi:10.1145/3122955.3122965.
- 50 Gordon D. Plotkin and Matija Pretnar. Handlers of algebraic effects. In Giuseppe Castagna, editor, *Programming Languages and Systems, 18th European Symposium on Programming, ESOP 2009, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009, York, UK, March 22-29, 2009. Proceedings*, volume 5502 of *Lecture Notes in Computer Science*, pages 80–94. Springer, 2009. doi:10.1007/978-3-642-00590-9_7.
- 51 Jon Postel. Transmission control protocol. *RFC*, 793:1–91, 1981. doi:10.17487/RFC0793.
- 52 Arjen Rouvoet, Casper Bach Poulsen, Robbert Krebbers, and Eelco Visser. Intrinsically-typed definitional interpreters for linear, session-typed languages. In Jasmin Blanchette and Catalin Hritcu, editors, *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2020, New Orleans, LA, USA, January 20-21, 2020*, pages 284–298. ACM, 2020. doi:10.1145/3372885.3373818.
- 53 Michele Sevegnani and Muffy Calder. Bigrapher: Rewriting and analysis engine for bigraphs. In *Computer Aided Verification - 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part II*, pages 494–501, 2016. doi:10.1007/978-3-319-41540-6_27.
- 54 Michele Sevegnani, Milan Kabác, Muffy Calder, and Julie A. McCann. Modelling and verification of large-scale sensor network infrastructures. In *23rd International Conference on Engineering of Complex Computer Systems, ICECCS 2018, Melbourne, Australia, December 12-14, 2018*, pages 71–81, 2018. doi:10.1109/ICECCS2018.2018.00016.
- 55 Rui Shi and Hongwei Xi. A linear type system for multicore programming in ATS. *Sci. Comput. Program.*, 78(8):1176–1192, 2013. doi:10.1016/j.scico.2012.09.005.
- 56 Robert E. Strom. Mechanisms for compile-time enforcement of security. In *Conference Record of the Tenth Annual ACM Symposium on Principles of Programming Languages, Austin, Texas, USA, January 1983*, pages 276–284, 1983. doi:10.1145/567067.567093.
- 57 Nikhil Swamy, Juan Chen, and Ravi Chugh. Enforcing stateful authorization and information flow policies in fine. In Andrew D. Gordon, editor, *Programming Languages and Systems, 19th European Symposium on Programming, ESOP 2010, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2010, Paphos, Cyprus, March 20-28, 2010. Proceedings*, volume 6012 of *Lecture Notes in Computer Science*, pages 529–549. Springer, 2010. doi:10.1007/978-3-642-11957-6_28.
- 58 Nikhil Swamy, Juan Chen, Cédric Fournet, Pierre-Yves Strub, Karthikeyan Bhargavan, and Jean Yang. Secure distributed programming with value-dependent types. *J. Funct. Program.*, 23(4):402–451, 2013. doi:10.1017/S0956796813000142.
- 59 Nikhil Swamy, Joel Weinberger, Cole Schlesinger, Juan Chen, and Benjamin Livshits. Verifying higher-order programs with the dijkstra monad. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16-19, 2013*, pages 387–398, 2013. doi:10.1145/2491956.2491978.

- 60 Wouter Swierstra. A hoare logic for the state monad. In *Theorem Proving in Higher Order Logics, 22nd International Conference, TPHOLs 2009, Munich, Germany, August 17-20, 2009. Proceedings*, pages 440–451, 2009. doi:10.1007/978-3-642-03359-9_30.
- 61 Bernardo Toninho and Nobuko Yoshida. Certifying data in multiparty session types. *J. Log. Algebraic Methods Program.*, 90:61–83, 2017. doi:10.1016/j.jlamp.2016.11.005.
- 62 Jesse A. Tov and Riccardo Pucella. Practical affine types. In *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26-28, 2011*, pages 447–458, 2011. doi:10.1145/1926385.1926436.
- 63 Philip Wadler. Linear types can change the world! In Manfred Broy, editor, *Programming concepts and methods: Proceedings of the IFIP Working Group 2.2, 2.3 Working Conference on Programming Concepts and Methods, Sea of Galilee, Israel, 2-5 April, 1990*, page 561. North-Holland, 1990.
- 64 Philip Wadler. Is there a use for linear logic? In Charles Consel and Olivier Danvy, editors, *Proceedings of the Symposium on Partial Evaluation and Semantics-Based Program Manipulation, PEPM'91, Yale University, New Haven, Connecticut, USA, June 17-19, 1991*, pages 255–273. ACM, 1991. doi:10.1145/115865.115894.
- 65 David Walker. *Advanced Topic in Types and Programming Languages*, chapter Substructural Type Systems, pages 3–43. The MIT Press, 2004.
- 66 Stephanie Weirich, Antoine Voizard, Pedro Henrique Avezedo de Amorim, and Richard A. Eisenberg. A specification for dependent types in haskell. *PACMPL*, 1(ICFP):31:1–31:29, 2017. doi:10.1145/3110275.