Formalizing Homogeneous Language Embeddings

Tony Clark tony.clark@tvu.ac.uk

Thames Valley University, St. Mary's Road, Ealing, London, W5 5RF, United Kingdom

Laurence Trat laurie@tratt.net

Bournemouth University, Poole, Dorset, BH12 5BB, United Kingdom

Abstract

The cost of implementing syntactically distinct Domain Specific Languages (DSLs) can be reduced by homogeneously embedding them in a host language in cooperation with its compiler. Current homogeneous embedding approaches either restrict the embedding of multiple DSLs in order to provide safety guarantees, or allow multiple DSLs to be embedded but force the user to deal with the interoperability burden. In this paper we present the μ-calculus which allows parameterisable language embeddings to be specified and analysed. By reducing the problem to its core essentials we are able to show how multiple, expressive, language embeddings can be defined in a homogeneous embedding context. We further show how variant calculi with user-defined safety criteria can be defined.

1 Introduction

Domain Specific Languages (DSLs) are mini languages used to aid the implementation of recurring problems. What identifies a particular language as being a 'DSL' is partly subjective; intuitively, it is a language with its own syntax and semantics, but which is smaller and less generic than a typical GPL such as Java. The DSL premise is simple: a one off, up front, cost allows classes of systems to be created at low cost and in a reliable and maintainable fashion [16].

DSLs have a long history, although they have often gone by different names [1]. Traditional, widely used, DSLs such as the UNIX `make` program and the `yacc` parsing system have been implemented as stand-alone systems, which are effectively cut-down programming language compilers and virtual machines rolled into one; the associated implementation costs and lack of practical reusability have hampered the creation of DSLs [10]. An alternative approach to stand-alone implementation is *embedding*, where a DSL is 'hosted' within a host programming language; in other words, the host languages' syntax is extended with the DSLs syntax. A simple example of such embedding is an SQL DSL; by using a DSL instead of an

external database library one gains several advantages such as the static detection of SQL syntax errors and the safe insertion of external values into SQL statements. Language extension has been a goal of language researchers for several decades (see e.g. [12]) but most early efforts were unable to prevent unintended and unwanted interactions between languages and their extensions [5]. Later approaches, though largely theoretical, did show that certain forms of language extension could avoid such problems [3,4]. More recently, DSL embedding approaches such as Stratego [2], XMF [6,7], Converge [17], Metalua [9], and others (e.g. [14,8,13]) have shown that this is a viable approach.

DSL embedding techniques can be classified as either *heterogeneous* or *homogeneous* [17]. Put simply, heterogeneous embedding (e.g. Stratego) is when the system used to compile the host language, and the system used to implement the embedding are different (note that this does not imply that the host language must be different than the language used to implement the embedding). In contrast, homogeneous embedding (e.g. Converge, Metalua, XMF) uses the language's compiler to compile the host language and to facilitate DSL embedding. Heterogeneous embedding has the advantage that it can be applied to any host language and any embedded language. However this means that heterogeneous embedding systems generally have little or no idea of the semantics of the languages they are embedding into, meaning that such techniques are hard to scale up [17]. Furthermore heterogeneous techniques typically assume that a single DSL is embedded into a single host language: multiple distinct DSLs must be manually welded together in order to create a single heterogeneous embedding which does not suffer from syntax errors. Homogeneous embedding, however, is inherently limited to a specific host language, but is typically able to offer greater guarantees about the safety of the resulting embedding, allowing larger and more complex DSLs to be embedded. Homogeneous embedding also places no conceptual restrictions on embedding multiple DSLs in one host language, or having DSLs interleaved within each other.

In practice, current homogeneous embedding technologies limit the extent to which multiple DSLs can be embedded without resorting to unwieldy hacks [9]. For example, Metalua allows multiple DSLs to be embedded within it, but requires manipulation of the global parser; no guarantees are made that different extensions will co-exist peacefully, or even that individual extensions are well-formed. Converge, on the other hand, allows multiple DSLs to co-exist and enforces reasonable safety guarantees, but does so by making DSLs unpleasantly syntactically distinct, and making embedding DSLs within each other extremely difficult.

We believe that the distance between the conceptual promise and current practical realities of homogeneous embedding are in large part because of a lack of understanding of the underlying theory of language embedding in a homogeneous setting. In this paper we present the μ-calculus for specifying and analysing homogeneous language embedding. The calculus extends the λ-calculus with facilities for defining and using language embeddings, allowing parameterisable language definitions to be scoped to portions of a source file, and to be nested arbitrarily

within each other.

This paper is structured as follows: Section 2 defines the syntax and semantics of the μ-calculus and shows how to precisely define an embedded language; Section 3 describes different categories of language embedding and how they are encoded in the calculus; Section 4 shows a how a language with more than one embedding can be defined using the calculus and how safety criteria are represented; finally, Section 5 concludes the paper with an analysis and discussion of further work.

2 μ: A Language Embedding Calculus

The μ-calculus is an extension of the λ-calculus that supports embedded languages. New languages can be added to μ in terms of a precise definition of their syntax, semantics and how they relate to the execution context of the host language.

The definition and subsequent use of an embedded language takes the form of a standard structure within the μ-calculus. This section defines the calculus and defines how languages are embedded within it. It is structured as follows: Section 2.1 defines the syntax of the calculus and provides an example of its use; languages are embedded in trms of their abstract syntax, Section 2.2 defines a data type that represents μ-calculus abstract syntax; Section 2.3 defines the semantics of the μ-calculus by embedding it within itself.

2.1 Overview

The syntax of the μ-calculus is:

E	::=		expressions
		V	variables
	$\mid$	$\text{fun}(V)E$	functions
	$\mid$	EE	applications
	$\mid$	$\text{if } E \text{ then } E \text{ else } E$	conditionals
	$\mid$	(E,E,E)	language definition
	$\mid$	$\text{lang } E : T[C]$	language embedding
T	::=	$\ldots$	syntax types
C	::=	$\ldots$	raw text

μ contains the conventional λ-calculus, plus language definition and language embedding components. A language definition defines a language's semantics as an interpreter and how to embed it in the context of a host.

A language embedding allows the use of a language within the host calculus. In essence, language definitions define interpreters and how to move from a host interpreter to the embedded language's interpreter. A language embedding is a use of the definition in a context provided by the host language.

More specifically, the language definition triple $(eval, load, unload)$ defines: an evaluator, $eval$, which evaluates the language in terms of its state; a loader, $load$,

that maps from the host language state to the embedded language state; and an unloader, *unload*, that translates the embedded state back into a host state. The following is an overview of how the calculus might be used:

```
// Define an embedded SQL-like language...
type SQL = ... // type definition for SQL.
let sql = (evalSQL,loadSQL,unloadSQL)
            where
              evalSQL = ...
              loadSQL = ...
              unloadSQL = ...
// Define an embedded HTML-like language...
type HTML = ... // type definition for HTML.
let html = (evalHTML,loadHTML,unloadHTML)
             where ...
// Use the two embedded language definitions.
// Perform database queries to produce all the
// (name,age) pairs for adults...
let results =
     lang sql:SQL[SELECT name,age from Customer WHERE age > 18]
in // Produce the HTML table showing the results...
   lang html:HTML[
     <TABLE>
       for name,age in results do
         <TR>
           <TD> name </TD>
           <TD> age  </TD>
         </TR>
     </TABLE>
   ]
```

2.2 Abstract Syntax Types

The type definition for the μ-calculus is as follows:

```
type Exp(T) =
       Var(String)
     | Lambda(String,Exp(T))
     | Apply(Exp(T),Exp(T))
     | If(Exp(T),Exp(T),Exp(T))
     | Lang(T)
```

The type definition is parameterized with respect to the type of embedded languages: `T`. If a single language L is embedded then the resulting type is `Exp(L)`. If more then one language, L and M, are embedded, then we use a disjoint type combinator to express the type of the resulting language: `Exp(L + M)`. Finally, a fix-point operator can be used to construct a type: `Y(Exp)` is the type of languages constructed by embedding the μ-calculus in itself.

2.3 Semantics

The semantics of the μ-calculus is defined as a language embedding as follows. The evaluator for the calculus can be any suitable definition. To maximise the potential for future extension we implement the evaluator as a state machine. This

ensures that any embedded language has access to all data *and* control structures of the language. The canonical state machine for a λ-calculus is the SECD machine [11]. The type definition is as follows:

```
type State = ([Value],Env,[Instr],State) | Empty
type Env = String->Value
type Instr = Exp(T) | App | If(Exp(T),Exp(T))
type Value = Basic | Closure | State
type Closure = (String,Env,Exp(T))
```

The evaluator is a state transition function. It is supplied with a current machine state, performs a single transition, producing a new state. It is also supplied with another state transition function `eval` to which it supplies the new state. By supplying `eval`, the basic μ-calculus evaluator can be extended:

```
evalExp(eval)(s) =
  case s of
    ([v],_,[],Empty) -> s
    (s,e,Var(n):s,d) -> eval(e(n):s,e,c,d)
    (s,e,Lambda(n,b):c,d) -> eval((n,e,b):s,e,c,d)
    (s,e,Apply(o,a):c,s) -> eval(s,e,a:o:App:c,d)
    (s,e,If(f,g,h):c,s) -> eval(s,f:If(g,h):c,d)
    (true:s,e,If(g,h):c,d) -> eval(s,e,g:c,d)
    (false:s,e,If(g,h):c,d) -> eval(s,e,h:c,d)
    ((n,e',b):v:s,e,App:c,d) -> eval([],e'[n->v],[b],(s,e,c,d))
    (R:():s,e,App:c,d) -> eval((s,e,c,d):s,e,c,d)
    (I:v:s,e,App:c,d) -> eval(v)
    ([v],_,[],(s,e,c,d)) -> eval(v:s,e,c,d)
  end
```

The semantics of the calculus defined above is standard except for the builtin operators: `R` and `I` which are used to *reify* and *intern* machine states. Assuming the existence of a parsing mechanism that indexes on the type of a language definition, the expression `lang(e,l,u):t[c]` is equivalent to the following expression:

```
I(newState)
  where newState = u(termState,initialState)
    where termState = e(startState)
      where startState = l(initialState,parse(t)(c))
        where initialState = R()
```

The expression above uses a parser that is indexed on the type of the embedded language. It is ouside the scope of this paper to analyse how parsing mechanisms can be supported by the μ-calculus; however, the parser produces values of the appropriate abstract syntax type.

The initial state is created by reifying the current μ-context. The initial state is supplied to the loader `l` together with the abstract syntax to produce a starting state for the embedded language evaluator. The starting state is supplied to the evaluator `e` to produce a terminal state. The terminal state along with the original initial state is supplied to the unloader to produce a new host language state. The new state is then interned by supplying it to the host language interpreter.

If we embed μ in itself then the load and unload operations are identity. Therefore the definition of μ is:

```
let Mu = Y(Exp)
let evalMu = Y(evalExp)
let loadMu((s,e,c,d),x) = (s,e,x:s,d)
let unloadMu(s,_) = s
let muL = (evalMu,loadMu,unloadMu)
```

Now we can write programs that arbitrarily nest the calculus in itself (given suitable sugarings for infix operators):

```
fun(x) lang muL:Mu[fun(y) lang muL:Mu[x + y]]
```

In conclusion a language definition consists of: a parser for the language (which is not considered further by this paper); a data type for the language abstract syntax; a context data type for the language evaluator; an evaluator that processes the context; a loader that maps from host contexts to embedded contexts; an unloader that maps from embedded contexts to host contexts.

3 Categories and Styles of Language Embedding

The μ-calculus can be used to embed any language *l* within a host *h*. The intended usage is that *h* is defined as a language within μ and then *l* is defined within *h*. The approach supported by μ forces a precise definition of *how l* is embedded within *h* including any safety criteria. μ allows the embedding to be analysed prior to implementation.

There are a number of different types of language embedding. Some embeddings are *functional* because uses of the language denote values; some are non-functional because they modify the host language context; many language embeddings require that the bindings from the host language are transferred to the embedded language; some embedded languages require private state and some require that the state can be communicated to other embedded languages.

The μ-calculus can be used to define what we term *uniform* or *ad-hoc* languages. Uniform languages are those that extend the μ-calculus interpreter, and thus allow languages to be embedded inside them using the standard μ-calculus techniques. Ad-hoc languages are those that define an arbitrary interpreter; while it is still possible to embed other languages within them, this must be done manually on a case-by-case basis. This section provides examples of different categories of language embedding using the μ-calculus.

3.1 A Simple Extension

One of the simple programming language-like features not found in μ is a *let-binding*. In this section we show how this can be added as a language embedding to μ. The type for expressions in the language `Let` is defined by extending the basic expression language:

```
type LetExp(T) = Let(String,Let(T),Let(T)) | Exp(T)
type Let = Y(LetExp)
```

An evaluator for `Let` is defined by extending the evaluator for the basic calculus:

```
let evalLetExp(eval)(s) =
  case s of
    (s,e,Let(n,x,b):c,d) -> eval(s,e,x:Let(n,b):c,d)
    (v:s,e,Let(n,b):c,d) -> eval([],e[n->v],[b],(s,e,c,d))
    else evalExp(eval)(s)
  end
```

The `Let` language definition follows the same structure as the `Mu` language definition: the load and unload operations are essentially identity mappings:

```
type Let = Y(LetExp)
let evalLet = Y(evalLetExp)
let loadLet((s,e,c,d),x) = (s,e,x:c,d)
let unoadLet(s,_) = s
let letL = (evalLet,loadLet,unloadLet)
```

Now, the let language can be used when it is embedded in the basic calculus which is now of type `Exp(Let)`:

```
fun(x) lang letL:Let[let y = x + 1 in y ]
```

Note also that because the `Let` language is uniform, it can be used as the basis for further language embeddings. This is shown in the following section.

3.2 Localized Data

A language extension is often useful when creating data structures. If the application domain is specialized then the language extension can provide abstractions that make the construction of the data *declarative* in the sense that low-level language features that are necessary to create the structure are hidden away.

The definition of a new feature for data is an example of a *functional* language embedding. Such a language is not necessarily uniform, however it does not modify the state of the host language and is used exclusively for its value.

This section provides an example of a functional embedding for implementing arrays. An array can be encoded in the μ-calculus using functions:

```
let mkArray(i) = null
let set(i,v,a) = if i = j then v else a(j)
```

There may be many initial values when an array is created. Without a declarative language feature to achieve this, the creation will involve many nested calls to `set`. The language `Array` is used to create arrays:

```
lang letL:Let[
  let mkArray = fun(limit) lang arrayL:Array [ 0 .. limit ]
  in mkArray(100)]
```

The abstract syntax of the array language is not an extension to `Exp`:

```
type ArrayExp = (ArrayVal,ArrayVal)
type ArrayVal = Var(String) | Int
```

The evaluator for the array language only requires information about the binding context from the host language. The evaluator creates a value of type `[Value]`:

```
let evalArray((lower,upper),env) =
```

```
letrec mkList(l,u) = if l = u then [l] else lower:mkList(l+1,u)
       deref(Var(n)) = env(n)
       deref(i) = i
in mkList(deref(lower),deref(upper))
```

The host language cannot manipulate values of type `[Value]` (and in general for an embedded language there may be a wide variety of 'foreign' values). Therefore the array unloader must translate between the array representation and the calculus representation:

```
let transArray : [Value] -> Closure
let transArray([],i) = (j,[],[| null |])
let transArray(v:a,i) =
  (j,a->transArray(a,i+i),[| if j = <i> then <v> else a(j) |])
```

The operation `transArray` defined above uses quasi-quotes to construct and manipulate abstract syntax in terms of concrete syntax. We will assume that this language feature is available in the μ-calculus since it is simply sugar for the equivalent expression in terms of AST constructors. Quasi-quotes have been implemented in a number of languages to support syntax manipulation including Template Haskell [15], Converge and XMF.

Given the translation from arrays (lists of values) to a closure-based representation, it is possible to define the array loader and unloader:

```
let loadArray((s,e,c,d),(l,u)) = ((l,u),e)
let unloadArray(a,(s,e,c,d)) = (transArray(a,0):s,e,c,d)
```

We can now define a language that embeds both the μ-calculus and arrays into `Let`

```
type Lang(T) = Exp(Let(Exp(T) + ArrayExp)
type Array = Y(Lang)
let arrayL = (evalArray,loadArray,unloadArray)
```

The language used in the example above is defined by `Exp(Array)`

3.3 State Modification

Not all languages are functional. A non-functional language can affect the state of the host language in some way. The impact of the language can be on the data, on the control flow or both. This section provides a simple ad-hoc language embedding that affects the state of data in the host language.

Consider the case where we want to print out a message each time a μ-calculus function is called. The calculus does not provide any features that allow us to toggle function tracing on and off. A new language feature is required that allows the following:

```
lang traceL:Trace[ traceOn ]
... // Tracing is now on...
lang traceL:Trace[ traceOff ]
... // No more tracing
```

We will assume that there are builtin functions called `enter` and `exit` in the μ-calculus that allow functions to be traced. So a function `fun(x) b` can be traced

by changing the function body to:

```
fun(x) exit(b,enter(x))
```

The language for tracing is very simple:

```
type Trace = traceOn | traceOff
```

When tracing is switched on, the embedded language makes a global change to the host state. Each closure and function expression must be modified to insert calls to the tracing functions:

```
let trace(s,e,c,d) = (trace(s),trace(e),trace(c),trace(d))
let trace(Empty) = Empty
let trace([]) = []
let trace(x:s) = tace(x):trace(s)
let trace(n->v) = n->trace(v)
let trace(n,e,b) = (n,trace(e),[| exit(<trace(b)>,enter(<n>)) |])
let trace(Var(n)) = Var(n)
let trace(Lambda(n,b) = Lambda(n,[| exit(<trace(b)>,enter(<n>)) |])
let trace(Apply(m,n)) = Apply(trace(m),trace(n))
let trace(App) = App
```

The `untrace` operator performs the changes in reverse. Now the tracing language can be defined in terms of the global modifier to the host language state:

```
let evalTrace(s,traceOn) = trace(s)
let evalTrace(s,traceOff) = untrace(s)
let loadTrace(s,t) = (s,t)
let unloadTrace(s,s') = s
let traceL = (evalTrace,loadTrace,unloadTrace)
```

The calculus with tracing in is defined by `Y(Lang)` where:

```
type Lang(T) = Exp(Exp(T) + Trace)
```

3.4 Control Flow

The previous section defines a non-functional embedding that influences the structure of data in the host language. Another form of non-functional embedding affects the control flow of the host language. This is only possible if the embedded language has access to the complete state of the host language. It can be achieved by passing continuations to the embedded language, however this makes it difficult to define transformations on the state. Instead, if evaluators are defined in terms of transition machines then embedded languages have access to the required information in an appropriate format.

Suppose that we want a new language construct that aborts the program under a given condition:

```
lang letL:Let[
  let x = f(100)
  in lang abortL:Abort[ stop if(x > 100) ]]
```

The condition under which the program aborts is written in the μ-calculus, therefore the language is defined as:

```
type Abort(T) = Exp(T)
```

```
type Lang(T) = Exp(Let(Exp(T) + Abort(Exp(T))))
```

Note that in the example above the type `Abort` is a synonym for `Exp`, however the parser will use a different constructor to tag the result of parsing the embedded language.

The evaluator for the embedded language must extend that of the μ-calculus:

```
let evalAbort(eval)(expState,trueState,falseState) =
  trueState    when eval(expState) = ([true],_,_,_)
  falseState   otherwise
```

The loader and unloader for the `Abort` language are defined as follows:

```
let loadAbort((s,e,c,d),x) = (([],e,[x],Empty),(s,e,c,d),([error],[],[],Empty)
let unloadAbort(s,_) = s
```

The language for `Abort` is defined:

```
let abortL = (evalAbort(Y(evalExp)),loadAbort,unloadAbort)
```

3.5 Private State

Previous examples have shown how the identifiers that are in scope within the host language can be passed down to an embedded language. In general, this is achieved by the loader for the language passing the current environment to the evaluator for the embedded language.

Multiple occurrences of the same embedded language may require access to shared data. This can be achieved through binding in the host language and making the variables in scope available to the embedded language. However, this is not always desirable since the embedded language must know the names of the variables that hold the values of its state. In general, it is unsafe to rely on the use of particular variable names to pass information from one language to another.

Another option is to encode the state required by the embedded language as part of the evaluation state of the host language. This requires that the host language state is extended. This section shows how the state is extended.

Consider a language `Secret` that has a single boolean flag. Each occurrence of the embedded language may choose to toggle the flag or print it out:

```
type Secret = Toggle | Print
```

There are two new elements of state required by the secret language: the flag; the stream of outputs. Therefore, the state of the host calculus becomes: `(s,e,c,d,f,o)` where `f` is a boolean flag and `o` is a sequence of boolean representing the output of the program.

```
type SecretState = ([Value],Env,[Instr],SecretState,Bool,[Bool])
```

We have already defined `evalExp` and do not want to change it to reflect the extended state. The solution is to wrap the definition of `evalExp` with a new definition that *lifts* the signature from `State -> State` to `SecretState -> SecretState`. This is defined as follows:

```
let evalExp'(eval)(s,e,c,d,f,o) = evalExp(eval')(s,e,c,d)
```

```
      where eval'(s,e,c,d) = eval(s,e,c,d,f,o)
```

The evaluator for `Secret` is simple:

```
let evalSecret((f,o),Toggle) = (!f,o)
let evalSecret((f,o,Print) = (f,f:o)
```

The language definition for `Secret` is:

```
let loadSecret((s,e,c,d,f,o),x) = ((f,o),x)
let unloadSecret((f,o),(s,e,c,d,_,_)) = (s,e,c,d,f,o)
let secretL = (evalSecret,loadSecret,unloadSecret)
```

The calculus with `Secret` embedded within `Let` can be defined as `Y(Lang)`:

```
type Lang(T) = Exp(Let(Exp(T) + Secret))
```

3.6 New Binding Schemes

The μ-calculus is statically scoped. Dynamic scoping allows variables to be bound to values such that they are available anywhere in the program during the evaluation of a given expression. Dynamic scoping is useful to capture the situation where a variable would need to be passed to many operations as an argument. Common Lisp is an example of a language that provides both static and dynamic scoping. The following example shows how a dynamic binding scheme works:

```
lang letL:Let[
  let add = fun(x) x + y
  in lang dynL:Dyn[dyn y = 100 in add(20)]]
```

The function `add` takes a formal parameter `x` and adds it to `y` which is currently not in scope. The embedded language `Dyn` binds `y` and then calls `add` supplying it with `20`. The result of the program is `120`.

The dynamic binding language requires a new type of environment for dynamic variables. Just as `Let` extends and contracts the static environment, `Dyn` extends and contracts the dynamic environment. However, the evaluator for the basic `Exp` language cannot reference a new type of environment since the state is fixed.

The solution is to introduce a new state element for a dynamic environment and to merge the static and dynamic environments when the `Dyn` interpreter passes control to the `Exp` interpreter. When the state is returned by the `Exp` interpreter, the dynamic environment is extracted and replaced into the `Dyn` state. The type `Dyn` is defined:

```
type Dyn(T) = DLet(String,Dyn(T),Dyn(T)) | Exp(T)
```

The evaluator for `Dyn` is:

```
let evalDyn(eval)(s) =
  case s of
    (s,e,y,DLet(n,x,b):c,d) -> eval(s,e,y,x:DLet(n,b):c,d)
    (v:s,e,y,DLet(n,b):c,d) -> eval([],e,y[n->v],[b],(s,e,y,c,d))
    ([v],_,_,[],(s,e,y,c,d)) -> eval(v:s,e,y,c,d)
    else evalExp'(eval)(s)
  end
```

When the evaluator `evalDyn` is called it checks for dynamic binding expressions at the head of the control. The dynamic binding expression evaluates the value-part and extends the dynamic environment in the machine state.

The evaluator `evalExp` must be *lifted* to take account of the extra state component:

```
let evalExp'(eval)(s,e,,y,c,d) = evalExp(eval')(s,y + e,c,d)
      where eval'(s,e,c,d) = eval(s',e',y,c,d')
        where (s',e',c,d') = (s,e,c,d)/y
```

The definition of `evalExp'` given above merges the dynamic and lexical environments when it calls `evalExp`. When the environments are merged. the lexical environment always takes precedence allowing lexically bound variables to shadow the dynamic variables. Since the lexical environment always shadows the dynamic environment, it is possible to remove the dynamic environment from the resulting state. The `/` operator removes the 'base' environment wherever it occurs in the supplied state.

3.7 Summary

This section has described categories of language embedding using the μ-calculus. The calculus can be used to design embedded language in terms of the semantics both of the language and its embedding within the host and each language definition takes the form of a triple: evaluator, loader and unloader. The design of an embedding must answer questions relative to the host and sibling languages: syntax; semantics; load and unload; safety criteria.

4 Example Application

This section provides an example of multiple embedded languages that can work together. When designing multiple embeddings we must consider the interaction of the languages and any safety criteria that prevent the languages interacting in undesirable ways. The μ-calculus approach explicitly represents components of the embeddings that make it easy to ensure safety criteria are achieved.

Consider writing web-applications that use relational database tables to store data and uses HTML to provide the user-interface. We will use the basic μ-calculus as the host language; it is representative of a general purpose host. Two languages are embedded within the host: SQL is used to process the database tables; HTML is used to produce the user-interface. An example of the use of this is given in section 2.1.

4.1 Database Queries

The first step is to define an SQL-like language. We limit this to selecting fields from a named database table where the field values satisfy a given predicate expression:

```
type SQLExp(T) = ([String],String,Exp(T))
```

The evaluator for SQL requires an extra state component that maps table names to tables. We represent tables as sequences of table rows:

```
type SQLState = ([Value],Env,[Instr],SQLState,Tables) | Empty
type Tables = String->DBTable
type DBTable = [DBRow]
type DBRow = String->Value
```

The evaluator must handle the extra SQL constructs:

```
let evalSQL(eval)(s) =
  case s of
    (s,e,(ns,n,b):c,d,t) -> eval(t(n)/ns:[]:s,e,sel(b):c,d,t)
    ((vs:vss):s,e,sel(b):c,d,t) ->
      eval([],e[ns->vs],[b],(vss:s,e,cif(vs):sel(b):c,d,t),t)
    (true:rs:vss:s,e,cif(vs):c,d,t) -> eval(vss:(vs:rs):s,e,c,d,t)
    (false:rs:vss:s,e,cif(vs):c,d,t) -> eval(vss:rs:s,e,c,d,t)
    else evalExp(eval')(s,e,c,d)
           where (s,e,c,d,t)    = s
                 eval'(s,e,c,d) = eval(s,e,c,d,t)
  end
```

The SQL evaluator defined above detects SELECT expressions at the head of the control, lookup the table in the environment t(n) and restruct the values to the named fields t(n)/ns. The machine then uses the new instructions sel and cif to process each value-tuple in turn and build up a sequence of values that satisfy the predicate expression b.

4.2 Web Page Generation

The HTML language is defined as follows:

```
type HTML(T) = [Row(T)]
type Row(T) = Row[Col(T)] | ForRow([String],String,Row(T))
type Col(T) = Col(Exp(T)) | ForCol([String],String,Exp(T))
```

The state for the HTML language uses a new state component that models the output:

```
type HTMLState = ([Value],Env,[Instr],HTMLState,[String]) | Empty
```

The evaluator is defined as follows:

```
let evalHTML(eval)(s) =
  case s of
    (s,e,[rs]:c,d,o) -> eval(s,e,:rs:tend:c,d,"<TABLE>":o)
    (s,e,tend:c,d,o) -> eval(s,e,c,d,"</TABLE>":o)
    (s,e,Row(cs):c,d,o) -> eval(s,e,cs:rend:c,d,"<TR>":o)
    (s,e,rend:c,d,o) -> eval(s,e,c,d,"</TR>":o)
    (s,e,Col(b):c,d,o) -> eval(s,e,b:cend:c,d,"<TD>":o)
    (v:s,e,cend:c,d,o) -> eval(s,e,c,d,"</TD>":v:o)
    (s,e,for(ns,n,r):c,d,o) -> eval(e(n):s,e,next(ns,r):c,d,o)
    (vs:vss:s,e,next(ns,r):c,d,o) -> eval(vss:s,e[ns->vs],r:c,d,o)
    ([]:s,e,next(ns,r):c,d,o) -> eval(s,e,c,d,o)
    else evalExp(eval')(s,e,c,d)
           where (s,e,c,d,o)    = s
                 eval'(s,e,c,d) = eval(s,e,c,d,o)
  end
```

The HTML evaluator defined above detects table declarations at the head of the control. HTML output is built up as each element of the table declaration is processed. The evaluator uses the new instructions tend, rend and cend to produce the terminating tags. The for-loops within rows and columns are processed using the new instructions for and next.

4.3 Language Collaboration and Safety Criteria

The state of the host language must merge the requirements of the two embedded languages. Therefore:

```
let evalExp'(eval)(s,e,c,d,t,o) = evalExp(eval')(s,e,c,d)
      where eval'(s,e,c,d) = eval(s,e,c,d,t,o)
```

The load and unload mappings for the language definitions can then peform the appropriate state projections:

```
let loadSQL((s,e,c,d,t,o),sql) = (s,e,sql:c,d,t)
let unloadSQL((s,e,c,d,t),(_,_,_,_,_,o)) = (s,e,c,d,t,o)
let loadHTML((s,e,c,d,t,o),html) = (s,e,html:c,d,o)
let unloadHTML((s,e,c,d,o),(_,_,_,_,t,_)) = (s,e,c,d,t,o)
```

The definition given above describes an idealized engine that processes a host language and two embedded languages: SQL and HTML. The information required by each language is maintained separately: t for database tables and o for the HTML output. The information from database tables can be made available to the web-output through the host language because bindings in μ are scoped over HTML loops.

Any *implementation* of μ, SQL and HTML is required to maintain the separation of concerns that define the safety criteria. Any language that supports language embedding must provide a mechanism that implements the state separation and must show that tables and HTML output can be processed in ways that are not consistent with their semantics as defined by the evaluators, loaders and unloaders.

5 Conclusions

In this paper we defined the μ-calculus which allows languages and language embeddings to be specified. We showed that the μ-calculus is sufficiently expressive that it can be used to add new language features to itself in a coherent fashion. We finally showed how the μ-calculus can be used to specify how DSLs such as an HTML generation lanaguge and SQL can be embedded within each other.

References

[1] J. Bentley. Programming pearls: little languages. *Communications of the ACM*, 29(8):711–721, Aug. 1986.

[2] M. Bravenboer and E. Visser. Concrete syntax for objects. Domain-specific language embedding and assimilation without restrictions. In D. C. Schmidt, editor, *Proc.*

OOPSLA'04, Vancouver, Canada, 2004. ACM SIGPLAN.

[3] L. Cardelli, F. Matthes, and M. Abadi. Extensible grammars for language specialization. In *Proc. Fourth International Workshop on Database Programming Languages - Object Models and Languages*, pages 11–31, Aug. 1993.

[4] L. Cardelli, F. Matthes, and M. Abadi. Extensible syntax with lexical scoping. Technical Report 121, Digital Systems Research Centre, Feb. 1994.

[5] H. Christiansen. A survey of adaptable grammars. *SIGPLAN Notices*, 25(11):35–44, Nov. 1990.

[6] T. Clark, A. Evans, P. Sammut, and J. Willans. An executable metamodelling facility for domain specific language design. In *Proc. 4th OOPSLA Workshop on Domain-Specific Modeling*, Oct. 2004.

[7] T. Clark, P. Sammut, and J. Willans. Beyond annotations: A proposal for extensible java (xj). In *Eighth IEEE Internal Conference on Source Code Analysis and Manipulation*, pages 229–238. IEEE Computer Society, 2008.

[8] K. Czarnecki, J. O'Donnell, J. Striegnitz, and W. Taha. DSL implementation in MetaOCaml, Template Haskell, and C++. 3016:50–71, 2004.

[9] F. Fleutot and L. Tratt. Contrasting compile-time meta-programming in Metalua and Converge. In *Workshop on Dynamic Languages and Applications*, July 2007.

[10] P. Hudak. Modular domain specific languages and tools. In *Proc. Fifth International Conference on Software Reuse*, pages 134–142, June 1998.

[11] P. J. Landin. The mechanical evaluation of expressions. *The Computer Journal*, 6(4):308–320, 1964.

[12] J. R. Metzner. A graded bibliography on macro systems and extensible languages. volume 14 of *SIGPLAN Notices*, pages 57–64, Oct. 1979.

[13] S. Seefried, M. Chakravarty, and G. Keller. Optimising Embedded DSLs using Template Haskell. In *Third International Conference on Generative Programming and Component Engineering*, pages 186–205, Vancouver, Canada, 2004. Springer-Verlag.

[14] T. Sheard, Z. el Abidine Benaissa, and E. Pasalic. DSL implementation using staging and monads. In *Proc. 2nd conference on Domain Specific Languages*, volume 35 of *SIGPLAN*, pages 81–94. ACM, Oct. 1999.

[15] T. Sheard and S. P. Jones. Template meta-programming for Haskell. In *Proceedings of the Haskell workshop 2002*. ACM, 2002.

[16] D. Spinellis. Reliable software implementation using domain specific languages. In G. I. Schuëller and P. Kafka, editors, *Proc. ESREL '99 — The Tenth European Conference on Safety and Reliability*, pages 627–631, Sept. 1999.

[17] L. Tratt. Domain specific language implementation via compile-time meta-programming. *TOPLAS*, 30(6):1–40, 2008.