# Extensible Grammars for Homogeneous Language Embedding

Tony Clark[1] and Laurence Tratt[2]

[1] Thames Valley University, St. Mary's Road, Ealing, London, W5 5RF, United Kingdom
tony.clark@tvu.ac.uk
[2] Bournemouth University, Poole, Dorset, BH12 5BB, United Kingdom laurie@tratt.net

**Abstract.** Homogeneous language embedding allows syntactically distinct Domain Specific Languages (DSLs) to be embedded in a host language in co-operation with its compiler. Although such embedded languages can define an entirely new parsing mechanism, they can extend the host languages parser, allowing DSLs and the host language to be arbitrarily interleaved. The $\mu$-calculus is a mechanism for formalising the definition of homogeneously embedded languages. Language embeddings can be semantics based or translational. We argue that a semantics based embedding is more abstract and provides a clear distinction between the embedded language and its host. The $\mu$-calculus supports both approaches and can be used to define and analyse language embeddings. We present a method for defining language embedding using the $\mu$-calculus and show how translational language embedding can be proved correct with respect to a semantics based definition.

## 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, 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 [20]. Although often referred to by different names, DSLs have been used in industry for many decades [2].

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 [11]. 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. [14]) but it is only relatively recently that approaches such as Stratego [4], XMF [5, 6], Converge [21], Metalua [9], and others (e.g. [19, 8, 18]) have shown that this is a viable approach.

DSL embedding techniques can be classified as either *heterogeneous* or *homogeneous* [21]. 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 can be applied to any host language and any embedded language; however such systems generally have little or no idea of the semantics of the languages they are embedding into, meaning that scaling up such techniques is difficult [21]. 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.

In contrast, homogeneous embedding 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 interleaving DSLs (and the host language) 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 a previous paper [7], we presented the $\mu$-calculus for specifying and analysing homogeneous language embedding. This paper extends the previously detailed core $\mu$-calculus (recapped in Section 4) with detailed mechanisms for specifying the parsing and standalone semantics of embedded languages in a way that is independent of any particular implementation technology. To show that this abstract approach can be made practical, we show how the formalism allows extensible parsers (Section 3) and the translation of embedded languages into host languages. We show via example how the extended $\mu$-calculus can be used to define the implementation specific embeddings presented in [4] (Section 5). We therefore believe that the extended $\mu$-calculus presented in this paper is the first system that is designed to support the formal definition of embedded language syntax and semantics.

## 2 Language Embedding

### 2.1 The problem

Consider the following program that mixes a command language with a language for expressing XML (in this case, HTML) output:

```
let limit = 10
in <TABLE>
    <TR>
      while i < limit {
        <TD> i </TD>
      }
    </TR>
   </TABLE>
```

To a seasoned programmer the intent of this program is likely to be obvious, even if the mechanism by which it is achieved is not. Execution of this program involves several transitions between the two languages. Initially, we are in a command language that establishes a local binding for the variable `limit`. The body of the local block performs a transition to the `XML`-based language in order to produce an HTML table. The table row is produced by making a further transition to the command language in order to enter a loop. The body of the loop makes a transition back to the `XML` language in order to produce each table entry.

The two languages have distinct syntax that can be defined separately. Each language has a semantics and we must describe an interface between the two languages that defines what happens when program evaluation makes a transition from one language to the other. Issues that must be addressed include: how do variables get passed from one language to another (for example `limit`)? how do values get returned from one language to another? can control flow in one language be affected by another language?

Fundamentally language embedding requires the existence of a host language and an embedded language. However these are not fixed roles, as shown in the above example where, at different points, both languages take the rôle of host or embedded language. Embedding can be achieved either by translating the host and embedded language into a common target language, or by defining a relationship between the independently defined semantics of both the host and the embedded language.

### 2.2 Our proposal

We believe that a translational approach is fundamentally less abstract than a semantic relationship. This is because a DSL that is defined in terms of a translation to a target language requires the semantics of the target language as part of its semantic definition. If the DSL is to be hosted in multiple languages then the semantics is not reusable: either the complete semantics must be redefined as a translation to the host language or the semantics of the embedded language relies on the existence of a translation from the host language to some common target.

Our proposed approach is based on all languages having an independent semantic definition. Each time a language is embedded in another language it must supply a relationship between the two semantics; the semantics are not modified in any way. Therefore, semantic definitions are reusable. Although separate semantic relationships must be defined for each embedding, such relationships are significantly simpler to define than semantic definitions.

A language definition has the following components:

**Abstract Syntax** A type definition for the data structure that represents the program.
**Concrete Syntax** A grammar that is processed by a parser to transform the language into a general parse tree format.
**Syntax Mapping** A mapping from a general parse tree format to abstract syntax for the language.
**State** A type definition for the semantic domain of the language.
**Operational Semantics** A function that executes the language. This paper gives some examples including a number of state transition machines.

A language embedding is a relationship between two language definitions. It defines how the execution context switches between the host-state and the embedded-state. A language embedding has the following components:

**Loader** A mapping from the host-state and the abstract syntax of the embedded language to an embedded-state.
**Unloader** A mapping from the embedded-state and a host-state to a new host-state.

The loader handles the evaluation context switch from host language to embedded language and the unloader handles the switch in the other direction.

### 2.3 The $\mu$-Calculus

The $\mu$-calculus is a language for expressing DSLs using the approach described above. The calculus can act as a host-language or an embedded-language. It provides some simple features that allow languages to be defined and the context switch to occur using the components described above.

The syntax of the core $\mu$-calculus is:

$$E ::= V \mid \lambda V.E \mid EE \mid (E,\ldots,E) \mid \mu E : E[Char*]$$

The calculus extends the $\lambda$-calculus with tuples (used for *language definitions*) and $\mu$-expressions which are *language embedding expressions*. A language embedding is an expression of the form: $\mu l : T[b]$ where $l$ is an expression denoting a language definition, $T$ is an expression denoting a syntax mapping, and $b$ is a sequence of characters delimited by [ and ] written in the concrete syntax of the language $l$. The context in which the language embedding expression occurs in the host language.

A language definition is a tuple: $((g,e),m,l,u)$ where $g$ is a grammar that describes the concrete syntax of the language, $e$ is an environment binding variables referenced in $g$, $m$ is an operational semantics for the language, $l$ is a loader for the language and $u$ is an unloader. For any given language, the elements $g$, $e$, $m$ and $T$ are fixed. The $l$

and $u$ components depend on the host language; therefore there may be many language definitions that embed a language in different hosts.

For practical purposes we will extend the core calculus with constructs that make working with languages easier. Global definitions are introduced by a `let`, $\lambda$ will be written `fun`, $\mu$ will be written `lang`. Simple type definitions will be used to describe data structures. Standard `case`-expressions with pattern matching over data will be used to describe data transformations including state transitions.

The following is a simple example, taken from [9], of the use of the calculus. A railway timetable consists of the times and destinations of trains that stop at a given station. Each train offers a mixture of premium and cheaply priced seats. We want an embedded language that can express timetable information declaratively. Suppose also that we want the timetable language to have access to variable values that are currently in scope. This will allow us to defined timetable templates that can be instantiated for different variable values. For example, the Salisbury train varies the minutes past the hour and the pricing for seats on the Waterloo train change in Summer and Winter:

```
lang railway:Railway[
  8:25   "Exeter St. Davids" Premium
  10:min "Salisbury"         Premium Cheap
  11:49  "London Waterloo"   WLOO_price
]
```

We can design an abstract syntax for this language:

```
type Railway     = [Train]
type Train       = (Time,Destination,[Price])
type Time        = (Exp,Exp)
type Destination = String
type Price       = Premium | Cheap | VarPrice(String)
type Exp         = Int(Integer) | Var(String)
```

The concrete syntax is defined using a grammar; the grammar language is not part of the core $\mu$-calculus, but is an embedded language (described in Section 3). The following is an example of a language embedding:

```
let railwayGrammar =
  lang grammar:Grammar[
    Railway     <- Train*.
    Train       <- t=Time d=Destination ps=Price {(t,d,ps)}.
    Time        <- h=Exp ':' m=Exp {(h,m)}.
    Destination <- Str.
    Price       <- 'Premium' {Premium}
                |  'Cheap' {Cheap}
                |  n=Name {VarPrice(n)}.
    Exp         <- i=Int {Int(i)}
                |  n=Name {Var(n)}.
  ]
```

The timetable language denotes a data structure that must be constructed in terms of an environment of variable bindings:

```
type RState = (Railway,String->Value)
evalRailway :: RState -> Value
let evalRailway(r,e)
  case r of
    v:vs     -> evalRailway(v,e):evalRailway(vs,e)
    []       -> []
    (t,d,ps) -> (evalRailway(t,e),d,evalRailway(ps,e))
    (h,m)    -> (evalRailway(h,e),evalRailway(m,e))
    Int(i)   -> i
    Var(n)   -> e(n)
    Premium  -> "p"
    Cheap    -> "c"
    VarPrice(n) -> e(n)
  end
```

The railway language is to be embedded within the $\mu$-calculus. As described later in this paper, the $\mu$-calculus has an operational semantics defined using an SECD machine. The states of the SECD machine are of type `State` and the elements are `(s,e,c,d)`. The load and unload components are defined as follows:

```
loadRail :: (State,Railway) -> RState
let loadRail((s,e,c,d),r) = (r,e)
unloadRail :: (Value,State) -> State
let unloadRail(timetable,(s,e,c,d)) = (timetable:s,e,c,d)
```

Finally we can define the Railway language:

```
let railway = (railwayGrammar,evalRailway,loadRail,unloadRail)
```

## 3 Syntax

The parsing of embedded languages can be performed by an external parser or by a parser that is integrated with the host language. Practically speaking, external parsers are either largely unaware of the relationship between the host language and the embedded language, or they have to duplicate many aspects of the host language compiler. Homogeneous language embedding seeks to integrate the embedded language as closely as possible, thus avoiding this problem.

The $\mu$-calculus provides a precise executable system for analysing language embeddings. This includes analysis of the syntactic relationship between the host language and the embedded language. A parser is specified in $\mu$ using a *grammar definition language* that is written as an embedded language definition in $\mu$. This section defines the grammar definition language and its semantics.

The concrete syntax of embedded languages are expressed using a grammar. Combining languages means combining parses, which is problematic using traditional parsing approaches which separate out tokenization (often called 'lexing') from parsing, since tokenization rules are rarely shared between languages. Scannerless parsing [17] avoids these problems by integrating the tokenization and parsing phases [22]. Our approach is thus based on scannerless parsing.

Although each language is defined using a separate grammar, the language used to express grammars and the parsing mechanism can be defined once and shared between

many languages. We use a synthesising parser that processes input with respect to a grammar and synthesises values of type `PValue` that are subsequently mapped to an abstract syntax value of the appropriate type.

### 3.1 General Syntax Values

The parsing mechanism synthesises a generic representation for abstract syntax values. The generic syntax tree type definition is as follows:

```
type PValue = Cnstr(String,[PValue])
            | Tuple[PValue]
            | Str(String)
            | Int(Integer)
```

A grammar is used to construct a value of type `PValue`. However, when parsing a language grammar we require a value of the appropriate abstract syntax type. Abstract syntax types are defined using a simple type language whose meta-language is `PValue`, so we can always automatically map any `PValue` onto a value of any type providing the tags (in `Cnstr`) match the tags in the type definition. We assume an operation `mapPVal` that performs this task.

### 3.2 Grammar Abstract Syntax

Grammars are represented as a value of type `Grammar` which is just a list of rules. Each rule has a name and a sequence of arguments. Arguments are supplied with values when rules are called by name within rule bodies. When a rule is called, the body of the rule is processed with respect to the input string in the context of the arguments. Body elements specify the format of the input string and can consume a prefix of it. Each body element synthesises a value of type `PValue` that may be bound to a local variable in the rule. A rule synthesises and returns the value produced by the last body element. Grammars are defined using the type definition in Table 3.2.

### 3.3 Grammar Concrete Syntax

The concrete syntax of grammars is defined as an embedded language. The definition uses the embedded language `Grammar` which means that this definition is recursive. Bootstrapping the parsing mechanism for the $\mu$-calculus is discussed later in this paper.

Grammars in the $\mu$-calculus can be composed. Composition allows common definitions to be factored out and reused by several different language definitions. The semantics of composition will be defined later. This section defines a collection of small general purpose grammars that are included by composition into various language definitions in the rest of the paper.

The following grammar provides some basic definitions:

```
let parseGeneral =
  lang grammar:Grammar[
    Name   <- Spaces c=Alpha cs=(Alpha | Num)* {c:cs}.
    Space  <- ' '.
```

```
type Grammar = [Rule]
type Rule    = (String,[String],Body)
type Body    = And(Body,Body)          process left then right
             | Or(body,Body)           process left or right
             | Bind(String,Body)       bind name to result
             | Try(Body)               try body without consuming
             | Star(Body)              greedily repeat, produce list
             | Call(String,[GExp])     call named rule with args
             | Literal(String)         match chars and consume
             | Range(Integer,Integer)  match char and consume
             | Not(Body)               body must not match
             | Test(GExp)              expression must be true
             | OK                      succeeds consumes nothing
             | Any                     succeeds consumes single char
             | Action(GExp)            evaluate exp, return value
type GExp    = Cnstr(String,[GExp])
             | Str(String)
             | Int(Integer)
             | Var(String)
```

**Table 1.** Grammar type definitions

```
    Spaces <- Space*.
    Alpha  <- ['a' - 'z'] | ['A' - 'Z'].
    Num    <- ['0' - '9'].
  ]
```

The rule `Name` specifies that any number of space characters are consumed, an alpha char is consumed and bound to the name `c`, any number of alpha-numeric chars are consumed and bound to the name `cs`. Finally, the rule returns a string which is represented by a sequence of characters `c:cs`.

Basic tokens are useful in a number of languages:

```
let basicTokens =
  lang grammar:Grammar[
    Lp    <- Spaces '('.
    Rp    <- Spaces ')'.
    Ls    <- Spaces '['.
    Rs    <- Spaces ']'.
    Colon <- Spaces ':'.
    Comma <- Spaces ','.
  ] + parseGeneral
```

Notice that the grammar `basicTokens` extends the grammar `parseGeneral`. The extension is achieved using the `+` operator. Informally the grammar composition operator merges the two grammars together producing a single grammar that contains both sets of rules. Grammar tokens are defined by extending the basic tokens:

```
let basicGrammar =
  lang grammar:Grammar[
```

```
    DefinedAs <- Spaces '<-'.
    Dot       <- Spaces '.'.
    Bar       <- Spaces '|'.
    Asterisk  <- Spaces '*'.
    OK        <- Spaces 'ok'.
    Bang      <- '!'.
    Hat       <- '^'.
    Lc        <- '{'.
    Rc        <- '}'.
    Quote     <- '\''.
    BackQ     <- '\\'.
    Query     <- '?'.
    Any       <- '$'.
    LitChars  <- !Quote c=Char cs=LitChars {c:cs}
               |  BackQ c=Char cs=LitChars {c:cs}.
  ] + basicTokens
```

The concrete syntax for grammars is defined as follows:

```
let grammarGrammar =
  lang grammar:Grammar[
    Grammar <- Rule*.
    Rule    <- n=Name a=Names DefinedAs b=Body Dot {(n,a,b)}.
    Body    <- l=And (Bar r=Body {Or(l,r)} | {l}).
    And     <- l=Bind (r=And {And(l,r)} | {l}).
    Bind    <- (Star | n=Name Eq b = Star {Bind(n,b)}).
    Star    <- a=Atom (Asterisk {Star(a)} | {a}).
    Atom    <- Call | Literal | Ok | Action
            |  Not | Test | Lp b=Body Rp {b}.
    Call    <- n=Name Hat es=Exps {Call(n,es)}.
    Literal <- Quote cs=LitChars Quote {Literal(cs)}.
    Ok      <- OK {OK}.
    Any     <- Any {Any}.
    Not     <- Bang Lp b=Body Rp {Not(b)}.
    Test    <- Query Lp e=Exp Rp {Test(e)}.
    Action  <- Lc Exp Rc.
  ] + grammarTokens
```

The grammar `grammarGrammar` includes a range of synthesis actions between braces. An action constructs values of type `PValue`. The actions can be constants, variable references, tuples or tagged-values.

### 3.4 Parse States

The semantics of an embedded language requires a semantic domain for the language. Where the embedded language is executable, we propose that a state machine is a suitable way of expressing the operational semantics. A state machine consists of a type definition for the machine states and a transition function that maps states to states. This section defines the state type for the grammar language.

A grammar machine is a parser whose states are of type `PState`:

```
type PState = State([PVal],PEnv,[PInstr],[Char],Cont,PState,Grammar)
            | Empty
```

```
type PEnv   = String->Pval
type PInstr = PBody(Body)
            | Bind(String)
            | Reset(String)
            | Fail
type Cont   = Cont([PVal],PEnv,Cont,[PInstr])
            | Empty
```

A state has the form `(s,e,c,t,k,f,g)`, where `s` is a sequence of p-values that are synthesized during a parse, `e` is an environment mapping local variable names to p-values, and `c` is the control which is a sequence of machine instructions. Typically the control contains abstract syntax trees which must be injected into the control type using a tag (`PBody`). To aid readability, tags on values in machine definitions will be omitted.

The concrete syntax string is element `t`. As the parse proceeds, the characters at the head of `t` are examined. Terminals in the grammar match and consume the appropriate prefix of `t`. A parser must provide a mechanism for dealing with the alternatives introduced by the `Or` body component and for dealing with variable scoping. Local scoping is handled by the continuation `k` which saves and restores the values of variables when a rule is called. Alternatives are handled by the failure continuation `f`. Finally, the parsing machine is used to process many different grammars, so the `g` component contains the rules that can be called.

### 3.5 Parsing Machine

Our approach to embedded languages requires a semantic definition that is independent of any context in which the language may be embedded. For an executable language this takes the form of a state transition function. The transition function is defined in the $\mu$-calculus using a `case`-expression to pattern match against the supplied state. Typically, a state transition function will be defined by case analysis on the head of the control. The operational semantics of the grammar language are as follows:

```
evalGrammar :: PState -> PState
let evalGrammar(s) =
  case s of
    (s,e,And(l,r):c,t,k,f,g)  -> (s,e,l:r:c,t,k,f,g)
    (s,e,Or(l,r):c,t,k,f,g)   -> (s,e,l:c,t,(s,e,r:c,t,k,f,g),f,g)
    (s,e,Bind(n,b):c,t,k,f,g) -> (s,e,b:Bind(n):c,t,k,f,g)
    (v:s,e,Bind(n):c,t,k,f,g) -> (v:s,e[n->v],c,t,k,f,g)
    (s,e,Try(b):c,t,k,f,g)    -> (s,e,b:Reset(t):c,t,k,f,g)
    (s,e,Reset(t):c,_,k,f,g)  -> (s,e,c,t,k,f,g)
    (s,e,Not(b):c,t,k,f,g)    -> ([],e,[b,Fail],t,Empty,(s,e,c,t,k,f,g),g)
    (s,e,Test(x):c,t,k,f,g)   ->
      (?:s,e,c,t,k,f,g) when e(x)
      f                 otherwise
    (s,e,Star(b):c,t,k,f,g)   -> (s,e,Or(next,skip):c,t,k,f,g)
      where
        next = And(Bind(n,b),And(Bind(ns,Star(b)),cons))
                 where cons = Action(Cnstr(':',[Var(n),Var(ns)]))
        skip = And(OK,Action(Cnstr('nil',[])))
        n and ns are new names
```

```
  (s,e,Call(n,es):,t,k,f,g) -> ([],ns->e(es),[b],t,k',f,g)
    where
      g  = _+[(n,ns,b)]+_
      k' = (s,e,c,k)
  (s,e,Literal(l):c,t,k,f,g) ->
    (l+s,e,c,t',k,f,g) when t=l+t'
    f                  otherwise
  (s,e,OK:c,t,k,f,g)          -> (?:s,e,c,t,k,f,g)
  (s,e,Any:c,i:t,k,f,g)       -> (i:s,e,c,t,k,f,g)
  (s,e,Fail:c,t,k,f,g)        -> f
  (v:_,_,[],t,(s,e,c,k),f,g) -> (v:s,e,c,t,k,f,g)
  (v:_,_,[],t,Empty,_,g)      -> v
end
```

The transitions are briefly described in turn. An `And` element is simply flattened at the head of the control. An `Or` element proceeds with the left alternative and adds the right alternative as a fail continuation. Binding involves performing the element and then extending the environment with the value found at the head of the stack. An element can be `Try`-ed providing that the input string `t` is reset when the element succeeds. A `Not` element causes failure if the element succeeds and success if it fails; this is achieved by adding an appropriate failure continuation and inserting the `Fail` instruction after the element `b`. A `Test` involves an expression `x`; we assume that `e(x)` evaluates the expression with respect to the environment. A `Star` element involves repeatedly performing `b` and producing a list of the synthesized values. A `Star` can be expanded to a grammar expression as shown in the transition. A `Call` looks the name of the rule up in `e`, binds the argument names to the evaluated argument values and makes a transition to the body `b`; notice how the success continuation `k` is used to save the current context that is restored when the rule returns. A `Literal` matches the prefix of the input `t` and consumes it; otherwise it fails. `OK` and `Any` succeed; the former produces an unknown result whereas the latter consumes a single character and returns it. The last two transitions show what happens then the control is exhausted.

### 3.6 Language Embedding

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 semantic relationship between two languages. The `eval` function provides an operational semantics of the language. The embedding consists of a loader and an unloader. The loader is responsible for mapping from the host state to an initial embedded state. The unloader is responsible for mapping from a terminal embedded state and the original host state to a new host state. The language embedding for the grammar language in the $\mu$-calculus is defined as follows (references to `State` are resolved in Section 4).

```
initialGrammar :: ((Grammar,String->Val),String) -> PState
let initialGrammar((g,e),t) = ([],e,[Call(start(g))],t,Empty,Empty,g)
let parse((g,e),t) = evalGrammar(initialGrammar(g,e,t)

loadGrammar :: (State,String) -> PState
```

```
let loadGrammar(_,t) = initialGrammar(grammarGrammar,t)

unloadGrammar :: (PState,State) -> State
let unloadGrammar((g:_,_,_,_,_,_,_,_),(s,e,c,d)) =  ((g,e):s,e,c,d)

let grammar = (grammarGrammar,evalGrammar,loadGrammar,unloadGrammar)
```

### 3.7 Bootstrapping a Parser

The grammar language `grammarGrammar` is defined as an embedded language that uses itself via the definition of `loadGrammar` given above. Consider the textual concrete syntax definition `grammarText` of `grammarGrammar`. The grammar `grammarGrammar` is a pair `(g,e)` for any environment `e` such that `g` is the smallest grammar that satisfies the following equation:

```
parse((g,e),grammarText) = g
```

## 4 $\mu$-Calculus

This section defines the $\mu$-calculus as a language embedded within itself. Each of the language definition components are defined in turn and then the complete language is given as a language embedding.

### 4.1 Abstract Syntax

The type definition for the $\mu$-calculus is as follows:

```
type Exp =
       Var(String)                 variables
     | Lambda(String,Exp)          functions
     | Apply(Exp,Exp)              applications
     | If(Exp,Exp,Exp)             conditionals
     | Tuple[Exp]                  tuples
     | Lang(Exp,Exp,Char*)         embeddings
```

### 4.2 Concrete Syntax

The concrete syntax of the $\mu$-calculus consists of some tokens and an expression grammar. The tokens `expTokens` are an extension to the `basicTokens` grammar defined earlier. The definition is omitted. The $\mu$-calculus grammar is defined as follows:

```
let expGrammar =
  lang grammar:Grammar[
    Exp     <- a=Atom Tail^(a)
             | Lambda
             | IfExp.
    Tail(a) <- b=Atom Tail^(Apply(a,b))
             |  {a}.
    Atom    <- n=Name {Var(n)}
```

```
          |  i=Numeric {Int(i)}
          |  Lparen e=Exp Rparen {e}.
          |  Lang p=Exp Colon l=Exp Ls t=Text Rs {Lang(p,l,t)}
          |  Tuple.
  Tuple   <- Lp e=Exps Rp {e}.
  Exps    <- e=Exp es=(Comma Exp)* {e:es}.
  Lambda  <- Fun Lp a=Name Rp e=Exp {Lam(a,e)}.
  IfExp   <- If t=Exp Then c=Exp Else a=Exp {If(t,c,a)}.
  Text    <- (!Rs $)*
 ] + expTokens
```

The $\mu$-grammar parses expressions. An expression is either an atom followed by an expression tail, a lambda function or an if-expression. Notice the use of `Tail` which uses the $\wedge$ operator to supply arguments (in this case `a`) to the rule. The rule `Tail` has an argument `a` and either builds a (left associative) application expression or just returns `a`. An atom is either a name, a number, a language-expression or a parenthesised expression. Functions and if-expressions use previously described constructs. A language expression is a sequence of characters (`$`), excluding (`!`) close square bracket.

### 4.3 Machine States

The semantics of the $\mu$-calculus itself can be defined as a language embedding. Although not the only way of defining the evaluator, we use an SECD machine [13], since it allows access to all data and control structures of the language.

The type definition is as follows:

```
type State   = MState([Value],Env,[Instr],State)
             | Empty
type Env     = String->Value
type Instr   = MInstr(Exp)
             | App
             | If(Exp,Exp)
             | Tup(Integer)
type Value   = Str(String)
             | Closure(String,Env,Exp)
             | State
             | Tuple[Exp]
             | Tagged(String,[Exp])
```

As for the parsing machine, the tags on values will be omitted where the meaning is clear from the context.

### 4.4 Operational Semantics

The evaluator is a state transition function. It is supplied with a current machine state, performs a single transition, and produces a new state:

```
evalExp :: State -> State
let evalExp(s) =
  case s of
    ([v],_,[],Empty)         -> s
```

```
    (s,e,Var(n):s,d)          -> (e(n):s,e,c,d)
    (s,e,Lambda(n,b):c,d)     -> ((n,e,b):s,e,c,d)
    (s,e,Apply(o,a):c,s)      -> (s,e,a:o:App:c,d)
    (s,e,If(f,g,h):c,s)       -> (s,f:If(g,h):c,d)
    (true:s,e,If(g,h):c,d)    -> (s,e,g:c,d)
    (false:s,e,If(g,h):c,d)   -> (s,e,h:c,d)
    (s,e,Tuple(x):c,d)        -> (s,e,x:Tup(#x):c,d)
    (vs:s,e,Tup(n):c,d)       -> (Tuple(vs):s,e,c,d) when #vs=n
    ((n,e',b):v:s,e,App:c,d) -> ([],e'[n->v],[b],(s,e,c,d))
    (R:():s,e,App:c,d)        -> ((s,e,c,d):s,e,c,d)
    (I:v:s,e,App:c,d)         -> (v)
    ([v],_,[],(s,e,c,d))      -> (v:s,e,c,d)
end
```

The semantics of the calculus defined above are standard except for the builtin operators `R` and `I` which are used to *reify* and *intern* machine states respectively. The expression `lang(g,e,l,u):t[c]` is equivalent to the following expression:

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

The initial state is created by reifying the current $\mu$-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 function `run` repeatedly applies the evaluator until a terminal state is reached). 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.

### 4.5 Language Embedding

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

```
loadExp :: (State,Exp) -> State
let loadExp((s,e,c,d),x) = (s,e,x:s,d)

unloadExp :: (State,State) -> State
let unloadExp(s,_) = s

let exp=(expGrammar,evalExp,loadExp,unloadExp)
```

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

```
fun(x) lang exp:Exp[fun(y) lang exp:Exp[x + y]]
```

In summary, a language definition consists of: a parser for the language; 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; and an unloader that maps from embedded contexts to host contexts.

## 5 Example Application

This section provides an example of multiple embedded languages that can work together. Embedded languages are often implemented using a translation to a target language. In many existing approaches, the translation *is* the semantics of the embedded language. The $\mu$-calculus approach separates the semantic definition of an embedded language from its implementation in terms of a translation. All aspects of the embedding are precisely defined and therefore it is possible to prove that a translation is correct.

The example is taken from [4] where a language for producing XML output is embedded in Java. It is beyond the scope of this paper to give a complete $\mu$-calculus definition of Java and XML so we make some simplifying assumptions by defining a small command language that is representative of Java and by defining a tagged tree language that is representative of XML. Our command language consists of while-loops, blocks containing local variables and print commands. One way of producing a simple XML document that contains a table with the numbers 0 to 99 arranged in a $10 \times 10$ grid is as follows:

```
lang commands:Commands[
  { i = 0;
    print("<TABLE>");
    while i < 10 {
      j = 0;
      print("<TR>");
      while j < 10 {
        print("<TD>");
        print((10 * i) + j);
        print("</TD>");
        j = j + 1;
      }
      i = i + 1;
      print("</TR>");
    }
    print("</TABLE>");
  }
]
```

The code above shows examples of all the commands supported by the simple language. A significant problem with this example is that although it implicitly uses two languages – the command language and XML – they can not be automatically distinguished. Since the XML is embedded as strings in print statements, the intent behind it is lost, and no guarantees can be made that all print statements relate to the XML language. This section shows how we can make the embedding of these languages explicit and precise.

### 5.1 A Command Language

The command language has the following syntax definition (the definition of `cmdTokens` is omitted):

```
let cmdGrammar = lang grammar:Grammar[
  Cmd  <- Lc d=Dec* c=Cmd* Rc {Block(d,c)}
```

```
        |  While e=Exp c=Cmd {While(e,c)}
        |  Print Lp e = Exp Rp Semi {Print(e)}
        |  n=Name Eq e=Exp Semi {Update(n,e)}
        |  Lang l=Name Colon t=Type Ls b=Text Rs {Lang(l,t,b)}.
  Exp  <- a=Atom (o=Op e=Exp {Bin(a,o,e)} | {a}).
  Atom <- Name | Str | Int.
  Op   <- '+' | ...
  Dec  <- n=Name Eq e=Exp Semi {Dec(n,e)}.
] + cmdTokens
```

The type definition for the command language is as follows:

```
type Cmd = Block([Dec],[Cmd])
         | While(Exp,Cmd)
         | Print(Exp)
         | Update(Name,Exp)
         | Lang(Name,Name,String)
type Exp = Bin(Exp,String,Exp)
         | Name(String)
         | Str(String)
         | Int(Integer)
type Dec = (Name,Exp)
```

The semantics of the command language is defined as a state machine. The states consist of an environment that maps names to strings and integers, a control that drives the machine, a dump that manages blocks and variable scoping and an output. The output is a string that is modified by print-commands.

```
type CState = (String->CVal,[Cmd],CState,String) | Empty
type CVal   = Int(Integer) | Str(String)

evalCmd :: CState -> CState
let evalCmd(s) =
  case s of
    (e,Block(ds,cs):c,d,o) -> (e,ds+cs,(e,c,d),o)
    (e,Dec(n,x):c,d,o)     -> (e[n->e(x)],c,d,o)
    (e,While(x,b):c,d,o)   -> (e,b:While(x,b):c,d,o) when e(x)
    (e,While(x,b):c,d,o)   -> (e,c,d,o)              when !e(x)
    (e,Print(x):c,d,o)     -> (e,c,d,e(x)+o)
    (e,Update(n,x):c,d,o)  -> (e[n->e(x)],c,d,o)
    (e,Lang(x,t,b):c,d,o)  -> (unload(terminal,s))
      where
        (g,eval,load,unload)     = e(x)
        syntax                   = mapPval(parse(g,b)e(t))
        start                    = load(s,syntax)
        terminal                 = eval(start)
  end
```

The state transitions use a definition of environment lookup, e.g. `e(x)` which generalizes name lookup to produce an expression evaluator. The main complexity arises in the transition that defines what happens when a `Cmd`-embedded language is performed. In this case, `Cmd` behaves in the same way as the $\mu$-calculus by unpacking the language definition, parsing the text and translating it into a syntax-tree, loading the syntax in the context of the `Cmd`-state `s`, evaluating the starting state and unloading the terminal state.

In order to embed this within the $\mu$-calculus we must describe how the states of the two languages correspond. The `Cmd` language is self contained except that the output component will persist between different embeddings. So in the following example, the output is `1,2,3,4,5,6,7,8,9,10`:

```
let printTo = fun(low,hi)
                lang cmd:Cmd[
                  while low <> hi {
                    print(low);
                    print(",");
                    low = low + 1;
                  }
                ]
in let step1 = printTo(1,5)
   in let step2 = printTo(6,10)
      in ...
```

To achieve this we must thread the output component through the host language state. The evaluator for `Exp` is modified to wrap the extra state component:

```
type State' = MState([Value],Env,[Instr],State,String)
            | Empty
evalExp' :: State' -> State'
let evalExp'(s,e,c,d,o) = (s,e,c,d,o)
  where (s,e,c,d) = evalExp(s,e,c,d)
let exp = (expGrammar,evalExp',loadExp,unoadExp)
```

Having introduced the new state component into the *mu*-calculus, the load and unload operations can be defined for the `Cmd` language:

```
loadCmd :: (State',Cmd) -> CState
let loadCmd((s,e,c,d,o),x) = (e,[x],Empty,o)

unloadCmd :: (CState,State') -> State'
let unloadCmd((_,_,_,o),(s,e,c,d,_)) = (s,e,c,d,o)

let cmd = (cmdGrammar,evalCmd,loadCmd,unloadCmd)
```

### 5.2 XML

The previous section showed an example where XML is produced by writing strings. This is unappealing because XML is a language whose processing rules and construction rules are not faithfully represented by strings. This section defines a cut-down XML language whose concrete syntax of XML is (omitting the definition of `XMLTokens`):

```
let xmlGrammar =
  lang grammar:Grammar[
    Tree      <- Element | Exp | Embed.
    Element   <- t = StartTag c=Tree* EndTag^(t) {Element(t,c,e)}.
    StartTag  <- Lc t=Name Rc {t}.
    EddTag(t) <- Lc Slash t'=Name Rc ?(t=t').
    Exp       <- a=Atom Tail^(a).
    Atom      <- Str | Var.
```

```
    Tail(l)   <- Add r=Exp Tail^(Add(l,r))
              | {l}.
    Embed     <- Lang n=Name Colon t=Name Ls b=Text Rs {Lang(n,t,b)}.
  ] + XMLTokens
```

Note the use of the test grammar expression `?(t=t')` that forces the terminal tag in an element declaration to match the start tag. The abstract syntax type definition is:

```
XML = Element(String,[XML])
    | Chars(Exp)
    | Lang(String,String,String)
Exp = Var(String)
    | Str(String)
    | Add(Exp,Exp)
```

The XML machine processes XML declaration and writes to an output string. The XML program may contain variables that are passed from the host language. Therefore, the state of the XML machine `(e,x,o)` contains an environment of variable bindings `e`, an XML program `x` and an output string `o`:

```
type XState = (String->XVal,XML,String)
type XVal   = Int(Integer) | Str(String)
```

The XML machine processes the tree and writes the data to the output. As in the `Cmd` language, `XML` allows other languages to be embedded:

```
evalXML :: XState -> XState
let evalXML(s) =
  case s of
    (e,Element(t,c):x,o)    -> (e,c+(Str(/t):x),t+o)
    (e,Str(s):x,o)          -> (e,x,s+o)
    (e,Var(n):x,0)          -> (e,x,e(n)+o)
    (e,Add(l,r):x,o)        -> (e,l:r:x,o)
    (e,Lang(d,t,b):x,o)     -> (unload(terminal,s))
      where
        (parse,eval,load,unload) = e(d)
        syntax                   = mapPval(parse(g,b)e(t))
        start                    = load(s,syntax)
        terminal                 = eval(start)
    (e,[],o)                -> s
  end
```

The `XML` language state requires a collection of bindings `e` and an output `o`. When `XML` is embedded within the $\mu$-calculus, it uses the same lexical environment. Like `Cmd`, `XML` uses a state component `o` that does not exist in the $\mu$-calculus. The output is a sequence of strings which is added to $\mu$ in exactly the same way as the output in `Cmd`, so the $\mu$-calculus machine state is modified in the same way as for `Cmd` in the previous section. Therefore, the language definition is:

```
loadXML :: (State',XML) -> XState
let loadXML((s,e,c,d,o),x) = (e,[x],o)

unloadXML :: (XState,State') -> State'
```

```
let unloadXML((_,_,o),(s,e,c,d,_)) = (s,e,c,d,o)

let xml = (xmlGrammar,evalXML,loadXML,unloadXML)
```

Give the changes to the $\mu$-calculus described above, the `XML` language can be embedded and used as follows:

```
let writeDoc = fun(content)
  lang xml:XML [
    <document>
       <header>
       </header>
       <body>
         content
       </body>
    </document>
  ]
```

The function named `writeDoc` is supplied with the content of an XML document. The function writes out boiler-plate code that wraps the content. Note that this is very different from writing out strings. The syntax of XML is checked and the start- and end-tags much match up.

### 5.3 Working Together

The aim of defining both the `Cmd` and the `XML` languages is to define an embedding for a declarative XML-based language within a standard imperative language. The previous sections have left the languages open so that other languages can be embedded within each of them. In practice, XML will be embedded within the command language in order to produce standard XML-templates. Within the XML-template, control will switch back to the command language.

Switching back and forth can occur arbitrarily and the relationship between the two languages must be defined. This is done by constructing new language definitions. The new language definitions are:

```
let cmd' = (parseCmd,evalCmd,loadCmd,unloadCmd)
             where
               loadCmd :: (XState,Cmd) -> CState
               loadCmd((e,_,o),c) = (e,[c],Empty,o)
               unloadCmd :: (CState,XState) -> XState
               unloadCmd((_,_,o),(e,x,_)) = (e,x,o)

let xml' = (parseXML,evalXML,loadXML,unloadXML)
             where
               loadXML :: (CState,XML) -> XState
               loadXML((e,_,_,o),x) = (e,[x],o)
               unloadXML (XState.CState) -> CState
               unloadXML((_,_,o),(e,c,d,_)) = (e,c,d,o)
```

The following example shows how the languages can be interleaved:

```
lang cmd:Cmd[
  { i = 0;
    lang xml':XML[
      <TABLE>
        lang cmd':Cmd[
          while i < 10 {
            j = 0;
            lang xml':XML[
              <TR>
                lang cmd':Cmd[
                  while j < 10 {
                    lang xml':XML[
                      <TD>
                        (10 * i) + j
                      </TD>
                    ]
                    j = j + 1;
                  }
                  i = i + 1;
                ]
              </TR>
            ]
          }
        ]
       </TABLE>
     ]
  }
]
```

The example above interleaves the two languages. The languages are always checked in terms of their syntax and the semantic relationship between the two languages is completely defined. If the language embeddings are fixed so that `XML` is the only language that can be embedded in `Cmd` and vice versa, then the syntax can be simplified to the following expression assuming that the host language is the $\mu$-calculus:

```
lang cmd:Cmd[
  { i = 0;
    <TABLE>
      while i < 10 {
        j = 0;
        <TR>
          while j < 10 {
            <TD>
              (10 * i) + j
            </TD>
            j = j + 1;
          }
          i = i + 1;
        </TR>
      }
    </TABLE>
  }
]
```

### 5.4 Translation

Section 5.2 defined a language `XML` that can be used to declare XML output in a general context. This section shows how XML can be embedded in a Java-like language (represented by the `Cmd` language) without modifying that language; in order to achieve this effect we translate the `XML` language into a collection of function calls. The following section addresses the issue of whether the translation is correct or not.

The translation is defined in terms of the abstract syntax as follows:

```
trans :: XML -> Cmd
trans(Element(t,c)) = Block([],[Print(Str(t))] +
                                  trans(c)     +
                                [Print(Str(/t))])
trans(Str(s))       = Print(Str(s))
trans(Var(n))       = Print(Var(n))
trans(Add(l,r))     = Block([],[trans(l),trans(r)])
trans(Lang(x,y,b))  = parse(CmdGrammar,b)
trans(x:xs)         = Block([],[trans(x),trans(xs)])
trans([])           = Block([],[])
```

The embedding of `XML` within `Cmd` can now be modified to reflect that we do not need the operational semantics of `XML` at all:

```
let xml' = (parseXML,evalCmd,loadXML,unloadXML)
             where
               loadXML :: (CState,XML) -> CState
               loadXML((e,c,d,o),x) = (e,trans(x):c,d,o)
               unloadXML :: (CState,CState) -> CState
               unloadXML(s,s') = s
```

The new definition is significantly changed. The semantics of embedding `XML` within `Cmd` now uses `evalCmd` after a translation from `XML` syntax to `Cmd` syntax.

### 5.5 Correctness

Sections 5.2, 5.1 and 5.3 have described a semantics-based embedding of a declarative language for XML within a standard command language. These definitions act as a specification of how the language should operate and how the language must behave when it is embedded.

When a DSL is embedded within an existing language, it is unusual for the host language to be modified in any way. Section 5.4 showed how the `XML` language can be translated into `Cmd` statements. In many systems, the translation *is* the semantics of the embedding. However, we have seen that an embedding can be expressed in terms of the source language which does not rely on the target language for its definition. By defining the embedding in terms of the semantics of the source language we have made a clear separation of concerns between the language and any number of possible embeddings in target languages. We have seen that `XML` has been defined once, but embedded within two languages: the $\mu$-calculus and `Cmd`.

There are two ways of embedding a language: *semantics based* (as defined in terms of load and unload operations) and *translation based* (as defined in the previous section). The semantics based embedding is more abstract since it is independent of the

target language. However, the translation based embedding is more practical since the target language is usually fixed and cannot be changed to accommodate the source language. The $\mu$-calculus can be used to specify a semantics based embedding and a translation. This section shows how the correctness of a translation with respect to the semantics can be established. The correctness criteria is expressed as follows:

```
unloadXML o evalXML o loadXML = evalCmd o loadTrans
  where loadTrans((e,c,d,o),x) = (e,trans(x):c,d,o)
```

The condition states that mapping a `Cmd` state to an `XML` state, performing, loading an `XML` program and then unloading back to a `Cmd` state is the same as translating the `XML` program to a `Cmd` program and performing it on the `Cmd` machine.

The proof is by induction on the structure of the `XML` program. We proceed by case analysis on the structure of `XML` programs:

```
case 1:
  (evalCmd o loadTrans)((e,c,d,o),Element(t,cs))
  = evalCmd(e,Block([],[Print(Str(t))]  +
                        trans(cs) +
                      [Print(Str(\t))]):c,d,o)
  = evalCmd(e,[Print(Str(t))]+trans(cs)+[Print(Str(\t))],(e,c,d),o)
  = evalCmd(e,trans(cs)+[Print(Str(\t))],(e,c,d),t+o)
```

At this point we assume by induction that the theorem holds for each element of `cs` and that the evaluation of `cs` produces some output `o'`:

```
  = evalCmd(e,[Print(Str(\t))],(e,c,d),o'+t+o)
  = evalCmd(e,c,d,\t+o'+t+o)
  = unload((e,[],\t+o'+t+o),(e,c,d,o))
  = (unloadXML(_,(e,c,d,o)) o evalXML)(e,[Element(t,cs)],o)
  = (unloadXML(_,s) o evalXML o loadXML)(s,Element(t,cs))
      where s = (e,c,d,o)
```

The following shows how variables are shown to be correct:

```
case 2:
  (evalCmd o loadTrans)((e,c,d,o),Var(n))
  = evalCmd(e,Print(Var(n)):c,d,o)
  = (e,c,d,e(n)+o)
  = unloadXML((e,[],e(n)+o),(e,c,d,o))
  = (unloadXML(_,(e,c,d,o)) o evalXML)(e,[Var(n)],o)
  = (unloadXML(_,s) o evalXML o loadXML)(s,Var(n))
      where s = (e,c,d,o)
```

Constants are the same as variables: The following shows how variables are shown to be correct:

```
case 3:
  (evalCmd o loadTrans)((e,c,d,o),Str(v))
  = evalCmd(e,Print(Str(v)):c,d,o)
  = (e,c,d,v+o)
  = unloadXML((e,[],v+o),(e,c,d,o))
  = (unloadXML(_,(e,c,d,o)) o evalXML)(e,[Str(v)],o)
  = (unloadXML(_,s) o evalXML o loadXML)(s,Str(v))
      where s = (e,c,d,o)
```

XML generates character output by concatenating expressions using +:

```
case 4:
  (evalCmd o loadTrans)((e,c,d,o),Add(l,r))
  = evalCmd(e,Block([],[trans(l),trans(r)]):c,d,o)
  = evalCmd(e,[trans(l),trans(r)],(e,c,d),o)
```

Assume by induction that:

```
(unloadXML o evalXML o loadXML)(e,[trans(l)],(e,c,d),o)
  = (evalCmd o loadTrans)(e,[trans(l)],(e,c,d),o)
```

and

```
(unloadXML o evalXML o loadXML)(e,[trans(r)],(e,c,d),o)
  = (evalCmd o loadTrans)(e,[trans(r)],(e,c,d),o)
```

Therefore:

```
  evalCmd(e,[trans(l),trans(r)],(e,c,d),o)
  = (e,[],c,d,o'+o''+o)
  = unloadXML((e,[],o'+o''+o),(e,c,d,o))
  = (unloadXML(_,(e,c,d,o)) o evalXML)(e,[l,r],o)
  = (unloadXML(_,s) o evalXML o loadXML)(s,Add(l,r))
      where s = (e,c,d,o)
```

We assume that the only language that can be embedded in XML is Cmd; therefore proving that Cmd can be translated to Cmd is true by definition. The case that proves a sequence of XML commands is the same as case 4 above. Therefore, all cases are covered and we have shown that the translation of XML to Cmd is correct with respect to the semantics of XML. QED.

## 6 Related work

Monticore [12] is a heterogeneous embedding approach that can be used to describe text-based languages and their relationship with UML class models. Monticore uses multiple, separate LL lexers and parsers (modifications of the ANTLR parsing tool [16]), which it switches between based on the presence of certain key tokens. Monticore is therefore able to gain the optimisation and familiarity of LL parsing tools. This unorthodox approach to parser composition raises several questions such as: is the resulting composition well formed? how flexible is the composition mechanism? Monticore is thus fundamentally different to the $\mu$-calculus parsing mechanism expressed in this paper which has a formal base, and uses the Packrat parsing mechanism.

MetaBorg [3] is a heterogeneous embedding approach which allows DSLs to be embedded within a GPL. Metaborg uses scannerless parsing to allow embedded DSLs to be parsed. It uses independent but co-operating code translations from the embedded language to the host language. In this way the semantics of the language extension is defined in the host language via a desugaring. This approach is possible with the $\mu$-calculus; however we offer a more abstract approach where each language embedding has a separate semantic definition.

*Rats!* [10] uses Packrat parsing to provide a modular parsing system. *Rats!* approach to parsing is fundamentally similar to that presented in this paper. The $\mu$-calculus can be seen as integrating a *Rats!*-like parsing approach into a formally defined language embedding system.

Converge [21] is a Python-like language which uses compile-time meta-programming (effectively macros) to homogeneously embed DSLs. Since it uses traditional non-scannerless parsing, Converge makes a harsh distinction between the host language and any embedded DSLs, making nested interleaving difficult. The $\mu$-calculus makes no such distinction, making nested interleaving no more difficult than top-level embedding. Converge however has a swathe of practical features relating to safety and error reporting that are not present in the core $\mu$-calculus, but which could be encoded. Metalua [9] is arguably the extant language closest in spirit to the $\mu$-calculus, as it allows parsing rules to be added to the parser at run-time, allowing DSLs to be interleaved. Unlike Converge and the $\mu$-calculus, however, the parsing mechanism it uses is unsafe and multiple DSLs can end up interacting in unanticipated and unwanted fashion.

## 7 Conclusions and Future Work

In this paper we defined the $\mu$-calculus which allows languages and language embeddings to be specified. We showed that the $\mu$-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 $\mu$-calculus can be used to specify how DSLs such as an HTML generation language and SQL can be embedded within each other.

XJ [6], Polyglot [15], and Fortress [1] are examples of language extension mechanisms for concrete programming languages. We intend using the $\mu$-calculus to analyse and compare the properties of different approaches. We also believe that the $\mu$-calculus is suited to formally defining and comparing module and language extension mechanisms such as [12] and [10].

## References


1. E. Allen, D. Chase, J. Hallett, V. Luchangco, J.-W. Maessen, S. Ryu, G. Steele, and S. Tobin-Hochstadt. The Fortress language specification. `http://research.sun.com/projects/plrg/-Publications/fortress.1.0.pdf`, 2008.
2. J. Bentley. Programming pearls: little languages. *Communications of the ACM*, 29(8):711–721, Aug. 1986.
3. M. Bravenboer, R. de Groot, and E. Visser. Metaborg in action: Examples of domain-specific language embedding and assimilation using stratego/xt. In *GTTSE*, July 2005.
4. 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.
5. 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.
6. 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.

7. T. Clark and L. Tratt. Formalizing homogeneous language embeddings, 2008. Submitted to LDTA 2008. http://itcentre.tvu.ac.uk/~clark/Papers/lang_embedding.pdf.
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. R. Grimm. Better extensibility through modular syntax. In *Proc. PLDI*, pages 38–51, New York, NY, USA, 2006. ACM.
11. P. Hudak. Modular domain specific languages and tools. In *Proc. Fifth International Conference on Software Reuse*, pages 134–142, June 1998.
12. H. Krahn, B. Rumpe, and S. Volkel. Monticore: Modular development of textual domain specific languages. In R. Richard Paige and B. Meyer, editors, *Proc. 46th International Conference Objects, Models, Components, Patterns (TOOLS-Europe). 2008.*, Zurich, Switzerland, 2008. Springer LNBIP.
13. P. J. Landin. The mechanical evaluation of expressions. *The Computer Journal*, 6(4):308–320, 1964.
14. J. R. Metzner. A graded bibliography on macro systems and extensible languages. volume 14 of *SIGPLAN Notices*, pages 57–64, Oct. 1979.
15. N. Nystrom, M. R. Clarkson, and A. C. Myers. Polyglot: An extensible compiler framework for java. In *In 12th International Conference on Compiler Construction*, pages 138–152. Springer-Verlag, 2003.
16. T. Parr. The reuse of grammars with embedded semantic actions. In *Proc. International Conference on Program Comprehension*, June 2008.
17. D. J. Salomon and G. V. Cormack. Scannerless nslr(1) parsing of programming languages. *SIGPLAN Not.*, 24(7):170–178, 1989.
18. 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.
19. 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.
20. 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.
21. L. Tratt. Domain specific language implementation via compile-time meta-programming. *TOPLAS*, 30(6):1–40, 2008.
22. E. Visser. Scannerless generalized-LR parsing. Technical Report P9707, Programming Research Group, University of Amsterdam, July 1997.