Creating Domain-Specific Languages by Composing Syntactical Constructs

Viktor Palmkvist and David Broman

KTH Royal Institute of Technology
Sweden
{vipa,dbro}@kth.se

Abstract. Creating a programming language is a considerable undertaking, even for relatively small domain-specific languages (DSLs). Most approaches to ease this task either limit the flexibility of the DSL or consider entire languages as the unit of composition. This paper presents a new approach using syntactical constructs (also called syncons) for defining DSLs in much smaller units of composition while retaining flexibility. A syntactical construct defines a single language feature, such as an `if` statement or an anonymous function. Each syntactical construct is fully self-contained: it specifies its own concrete syntax, binding semantics, and runtime semantics, independently of the rest of the language. The runtime semantics are specified as a translation to a user defined target language, while the binding semantics allow name resolution *before* expansion. Additionally, we present a novel approach for dealing with syntactical ambiguity that arises when combining languages, even if the languages are individually unambiguous. The work is implemented and evaluated in a case study, where small subsets of OCaml and Lua have been defined and composed using syntactical constructs.

1 Introduction

Designing and implementing user friendly *domain-specific languages (DSLs)* requires both extensive programming language knowledge and domain expertise. Instead of implementing a DSL compiler or interpreter from scratch, there are several approaches for developing new DSLs more efficiently. For instance, a language can be defined by compiler construction [8] or preprocessing template tools [3, 20] that translate a DSL program into another language with well defined syntax and semantics. Another alternative is to *embed* [14] the DSL into another host language, thus reusing the language constructs directly from the host language. Such an approach, often referred to as *embedded DSLs*, has been used in various domains [1, 2, 11, 24, 27].

New language constructs can, for instance, be implemented using host language constructs that lifts programs into data [4,19], or by using various forms of *macro systems* [23]. A macro defines a new construct by expanding code into the host language, thus giving the illusion of a new language construct without the need to redefine the underlying language. The sophistication of a macro system varies from simple text expansion systems to systems using hygienic macros that enable correct name bindings [6,9] and macro systems with static types [13,16].

Macro systems enable rapid prototyping of language constructs, but the concrete syntax of a macro tends to be limited to the syntax of the language, e.g., Lisp macros look like Lisp. On the other hand, compiler construction tools, such as parser generators and transformation frameworks, enable a higher degree of flexibility in terms of syntax, but do not directly give the same composability properties; the smallest unit of reuse tends to be a language, in contrast with macros, which are more fine-grained.

In this paper, we develop the concept of *syntactical constructs* (also called *syncons* for short) that enables both *composable language constructs* and *syntactic freedom*. In contrast to current state-of-the-art techniques, the composability is fine-grained, at the language construct level. That is, instead of composing complete DSLs, syncons enable composability of individual language constructs. A syncon defines a single language feature, such as an `if` statement or an anonymous function. Each syncon specifies its own syntax, binding semantics, and runtime semantics, independently of the rest of the language. The semantics are defined using a translation into another target language, similar to macros.

However, fine-grained composability introduces further challenges regarding unambiguous parsing. For instance, composing two individual language constructs picked from two different languages may create an ambiguous language as a result, even if the two languages are individually unambiguous. The approach presented in this paper uses general context-free grammars for syntactical flexibility, which presents a problem: static ambiguity checking for context-free grammars is undecidable [5]. A novel aspect of our approach is dynamic ambiguity checking, which means that errors are encountered and reported at parse time, similarly to how dynamically typed languages present type errors at runtime.

As part of our work, we have designed and implemented a prototype system for creating languages using the *syncon approach*. We make no assumptions about the target language, but for the purposes of evaluation, we have implemented a small interpreted language to fill this role. Fig. 1 shows a high-level overview of our approach, where the different sections (2-4) are highlighted with dashed lines. More specifically, we make the following main contributions:

– We explain the key idea of the *syncon* concept, as well as how the concrete syntax of a composed language is constructed. All parsing operations are performed by first constructing a context-free grammar, which is then handed to a general parser (Section 2).
– We motivate why dynamic ambiguity checking is, in some cases, preferable to fully unambiguous languages, and explain our approach. Specifically, when an ambiguity is encountered, the parser produces multiple parse trees, which are examined in order to present a useful error message (Section 3).
– We implement name resolution and expansion (Section 4), and evaluate the whole approach using a case study, where small subsets of OCaml and Lua are defined using syncons. We show how language constructs in one language can be extracted and composed into the other language (Section 5).

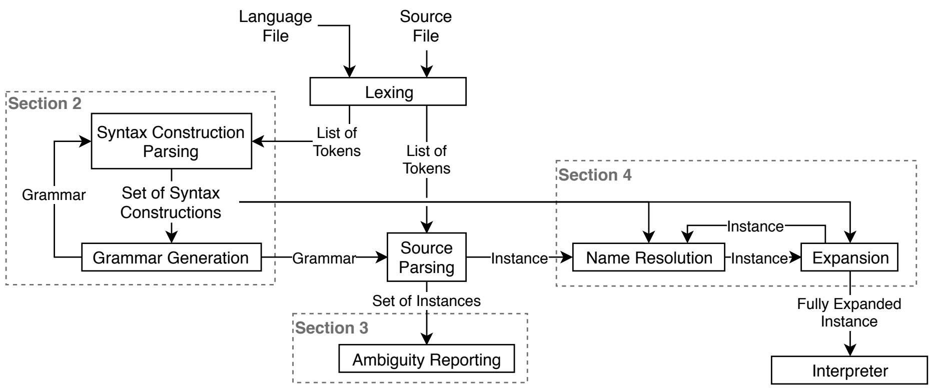

Fig. 1. An overview of the various components of the system, and the sections that explain them.

2 Defining Syncons

The central component in the approach presented in this paper is a *syncon*, short for syntactical construct. Each syncon belongs to a single *syntax type*, similar to how a value in a regular programming language belongs to a type. Fig. 2a shows an example that defines a slightly simplified version of a local variable declaration in Lua. The definition consists of three parts: a header, a set of properties, and a body.

2.1 Header

The header (lines 1 through 3 in Fig. 2a) contains three things: the name of the syncon (`local`), the syntax type to which it belongs (`Statement`), and a syntax description (lines 2 and 3). `Statement` and `Expression` are user-defined syntax types that define the syncon's relation to the target language.

The *syntax description* of a syncon describes its concrete syntax. It is similar to a production in Extended Backus Naur form (EBNF): it is a sequence of quoted literals, syntax types (non-terminals), and repetitions (via `?` for zero or one, `*` for zero or more, or `+` for one or more). Parentheses are used for grouping and have otherwise no effect on the described syntax.

The context-free grammar generated for parsing has one non-terminal per user-defined syntax type, and one production (in EBNF) per syncon. Quoted literals and a few builtin syntax types (`Identifier`, `String`, `Integer`, and `Float`) are terminals.

2.2 Properties

The properties of `local` (lines 5 and 6 in Fig. 2a) specify its binding semantics, i.e., which names it introduces, and how they are available to other code. A syncon can specify its binding semantics in two essentially orthogonal ways:

```
 1  syntax local:Statement =      Header
 2    "local" x:Identifier
 3    ("=" e:Expression)?
 4  {
 5    #bind x after              Properties
 6    #scope (e)
 7    BExpression‘                     Body
 8      defAfter ‘id(x) =
 9      (@ref (@deref
10        ‘t(foldl e _ (e)
11             (BExpression‘ @unit))))
12  }                                    (a)
```

```
1  if true then
2    local a
3    local b = 1 + 2
4    return b
5  else
6    return 3
7  end                (b)
```

```
syntax if:Statement =
  "if" c:Expression
  "then" b:Block
  ... "end"
{ #scope (b) ... }    (c)
```

Fig. 2. (a) A syncon implementing a basic form of a local variable declaration in Lua. (b) Code in Lua with two local declarations that can be parsed using this syncon. (c) A syncon implementing `if`, with most details elided.

- As an *adjacent* binding; for instance, `#bind x before` or `#bind x after`. This binds the identifier `x` in code appearing before or after the current syncon, respectively. Line 5 of Fig. 2a states that `x` (from the syntax description on line 2) is available only after the end of the local declaration. For example, `b`, introduced on line 3 in Fig. 2b, is bound on line 4, but not on line 2 or 3. The extent of an adjacent binding can be limited by a parent syncon specifying a scope; for instance, `#scope (e1 e2)`. This ensures that no adjacent binding in subtrees `e1` or `e2` can be seen from the outside, while allowing both `e1` and `e2` access to the bindings introduced in the other. The `#scope` declaration in Fig. 2c ensures that no bindings introduced in the **`then`** branch are accessible outside it.
- As a *nested* binding; for instance, `#bind x in e`. This binds identifier `x` in the subtree represented by `e`.

Fig. 3 shows the AST for the code in Fig. 2b. The dashed boxes denote the regions covered by **`before`** and **`after`** in **`local`** `b` on line 3 in Fig. 2b. The region **`in`** shows which regions *could* be covered by a `#bind x in e` declaration. The horizontal bars represent scopes, which limit the extent of the adjacent bindings, showing that **`if`** introduces a scope around each of its two branches. Had these *not* been there, then **`before`** would have included **`true`** as well, while **`after`** would have included the right-most `block` and all its descendants (i.e., the **`else`** branch) as well.

Note that a single syncon may use any number of adjacent and nested binding declarations, though usually with different identifiers. Adjacent bindings are, to the best of our knowledge, novel, and give two advantages over purely nested bindings:

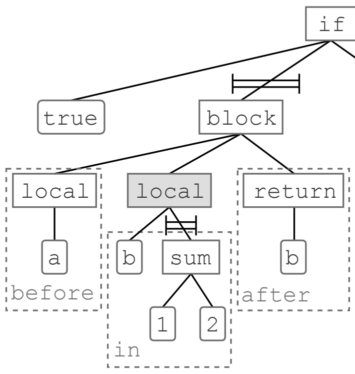

Fig. 3. The AST of the code in Fig. 2b.

1. Syncons may introduce bindings that can be used in a mutually recursive way. For example, a pair of mutually recursive functions require the other function to be in scope in its own body, but purely nested bindings can only accomplish this for one of the functions.
2. The binding constructs common in imperative languages can be modeled more simply. For example, the `local` declaration defined in Fig. 2a does not contain the statements that follow it, which would be required if only nested bindings were available.

The remaining properties are associativity and precedence, similar to most parser generators, that transform the generated context-free grammar appropriately.

2.3 Body

The body of `local`, found on lines 7 through 11 in Fig. 2a, specifies how it is translated to the underlying target language. The expansion is specified in a small DSL with three kinds of constructs: variable references, folds, and syntax literals. Variables are most commonly introduced in the syntax description in the header (`x` and `e` on lines 2 and 3 in Fig. 2a, respectively). Folds are used to reduce a sequence of syntax trees to a single syntax tree. For example, the fold on lines 10 and 11 in Fig. 2a has the following form:

```
foldl e _ (e) (BExpression‘ @unit)       │ List.fold_left
      ▲ ▲  ▲    ▲                        │   (fun _ e -> e)
      │ │  │    initial accumulator value │   ()
      │ │  fold function                 │   e
      │ accumulator name                 │
      list and element name              │
```

The OCaml code to the right has the same meaning, but with more familiar syntax. It is a left-fold, folding over the sequence `e` (which has length 0 or 1, since it was introduced by ?). The accumulator is given the name "_" (i.e., it is ignored), while the current element in the sequence reuses the name e. The folding function is merely e, i.e., it ignores the accumulator and returns the element as is. The initial value of the accumulator is a @unit value, which models Lua's **nil**. The end result is e if it is non-empty, and @unit otherwise. If * or + is used in the syntax description, folds can also be done over longer lists.

A syntax literal is introduced by a syntax type and a backtick (BExpression‘ on lines 7 and 11 in Fig. 2a) followed by code with that syntax type. Code to be run at expand-time can be spliced into a literal using one of several forms: ‘**t**() for a user-defined syntax type (line 10 in Fig. 2a), ‘**id**() for an identifier (line 8 in Fig. 2a), ‘str() for a string, ‘int() for an integer, and ‘float() for a float.

BExpression is a syntax type in the target language, while defAfter, @ref, @deref, and @unit are constructs in that language. The workings of these are not relevant to the syncon approach, and are thus omitted.

The need to specify the syntax type of the syntax literal stems from a lack of context. The issue is that the meaning of some pieces of syntax depend on their context, even in a context-free language. For example, in OCaml,

```
let example [1] = [1]
```

contains the syntax "[1]" twice, first as a pattern, then as an expression. In a syntax literal this context is absent, thus we require the language implementer to specify the syntax type using, e.g., BExpression‘. Similarly, each spliced expression must be tagged by syntax type.

3 Ambiguity Reporting

This section first argues that a dynamic check is in some cases preferable to an unambiguous grammar, which is followed by an explanation of our approach.

3.1 Motivation

Consider the following nested **match**-expression in OCaml:

```
1  let result = match 1 with
2    | 1 -> match "one" with
3           | str -> str
4    | 2 -> "two"
```

The compiler reports a type error, stating that line 4 matches a value of type int, but expects a value of type string. The compiler sees the **match**-arm on line 4 as belonging to the **match** on line 2, rather than the one on line 1, which is what the programmer intended. This happens because OCaml has no layout rules—the indentation has no impact on the semantics of the program—and the compiler assigns every **match**-arm to the closest **match**, resulting in a type error. Instead, the appropriate solution is to surround the inner **match**-expression with parentheses, which has nothing to do with the types of the patterns. Dynamic ambiguity checking can instead detect this case as an ambiguity and present it as such, yielding a clearer error message. We return to this example in the evaluation in Section 5.

3.2 Finding Ambiguities

In the case of ambiguous source code, the parser will produce multiple parse trees, a so called *parse forest*. This is a programmer error, so the system must produce a useful error message. In particular, it is insufficient to merely say ”the source code is ambiguous” since the ambiguity likely involves a very limited portion of the code. Additionally, once we have isolated the truly ambiguous portions of the source code, we must present the different interpretations in an understandable way. This subsection considers the former problem, while Section 3.3 deals with the latter.

To aid in our discussion, consider the following parse forest, produced by parsing ”`1 + 2 * 3 * 4`” when we have defined precedence but not associativity:

$$\texttt{add}_{1:7}(1_1, \texttt{mul}_{3:7}(2_3, \texttt{mul}_{5:7}(3_5, 4_7))) \quad \Big| \quad \texttt{add}_{1:7}(1_1, \texttt{mul}_{3:7}(\texttt{mul}_{3:5}(2_3, 3_5), 4_7))$$

The subscripts signify the source code area covered (henceforth referred to as the *range* of the parse tree). We see that the two parse trees share some structure: both have the form $\texttt{add}_{1:7}(1_1, \texttt{mul}_{3:7}(\_, \_))$. The addition is thus unambiguous and we wish to report only the multiplication as ambiguous. In particular, we can find a parse forest for the range $3 : 7$ whose parse trees appear as descendants in the full parse forest:

$$\texttt{mul}_{3:7}(2_3, \texttt{mul}_{5:7}(3_5, 4_7)) \quad \Big| \quad \texttt{mul}_{3:7}(\texttt{mul}_{3:5}(2_3, 3_5), 4_7)$$

We will refer to such a parse forest as a *subforest*.

Finding an ambiguity is now equivalent to finding a subforest whose parse trees differ, while the full ambiguity report selects a set of such subforests.

We require three helper functions before the actual algorithm. First, we will consider two parse trees to be shallowly equal (written $=_s$) if they only differ in their children (or do not differ at all). For example, $\texttt{add}_{1:3}(1, 2) =_s \texttt{add}_{1:3}(4, 5)$, but $\texttt{add}_{1:4}(1, 2) \neq_s \texttt{add}_{1:3}(4, 5)$ (different range) and $\texttt{add}_{1:3}(1) \neq_s \texttt{add}_{1:3}(1, 2)$ (different number of children). Second, $\mathit{children}(t)$ denotes the children of the syntax tree t, while $\mathit{children}(t)_i$ is the ith child of t, going left to right. Third, $\mathit{range}(t)$ is the range of the syntax tree t. The algorithm involves three steps, starting with the complete parse forest as the input F:

```
function AMBIGUITIES($F$)
    if $\exists t_1, t_2 \in F.\ t_1 \neq_s t_2$ then
        return $\{F\}$
    end if                                                        Step 1
    If all $t \in F$ are shallowly equal, then they have
    the same number of children. Call this number $n$.
    $S \leftarrow \{\{children(t)_i \mid t \in F\} \mid 1 \leq i \leq n\}$
    if $\exists S' \in S.\ \exists t_1, t_2 \in S'.\ range(t_1) \neq range(t_2)$ then
        return $\{F\}$
    end if                                                        Step 2
    return $\bigcup_{S' \in S}$ AMBIGUITIES(S')                     Step 3
end function
```

The first step checks that all trees are shallowly equal, the second step extracts all direct subforests, and the third recurses on those subforests. If either of the first two steps fail, then we can find no smaller subforest for the current range, thus we return F.

Applying this to the example, we quickly find that the forest only contains trees that are shallowly equal (step 1), and it has two subforests: $\{1_1\}$ and $\{\texttt{mul}_{3:7}(2_3, \texttt{mul}_{5:7}(3_5, 4_7)), \texttt{mul}_{3:7}(\texttt{mul}_{3:5}(2_3, 3_5), 4_7)\}$ (step 2). The former is trivially unambiguous, but the latter is not: while the trees are shallowly equal (step 1), we cannot extract subforests (step 2), since the children do not cover the same range.

3.3 Reporting Ambiguities

With the ambiguities found, in the form of subforests, they must be presented to the user. Each contained parse tree could be arbitrarily large, thus presenting them in their entirety is likely to be more noise than valuable information. It is our hypothesis that merely presenting two levels of the trees (the root and its children) is sufficient information to begin addressing the problem and includes very little noise.

For example, the ambiguity in the previous section is presented as follows:

$$\texttt{mul}_{3:7}(\texttt{integer}_3, \texttt{mul}_{5:7}) \quad \Big| \quad \texttt{mul}_{3:7}(\texttt{mul}_{3:5}, \texttt{integer}_7)$$

Furthermore, the range information can be used to highlight the corresponding regions of the source code.

As a final point, in the presence of a grouping operation (e.g., parentheses) ambiguities involving operators can be presented in a more natural way, even if our prototype implementation does not yet support it.

4 Binding Semantics and Expansion

Once we have parsed a syntactically unambiguous program and produced an AST we now turn our attention to its semantics, in particular, its binding semantics. The name resolution pass (Section 4.1) discovers the connections between binders and bound identifiers, as well as any binding errors that may be present, while expansion (Section 4.2) transforms the AST in the parsed language to an AST in the target language.

4.1 Name Resolution

The name resolution pass is implemented as a relatively simple tree-traversal that collects adjacent and nested bindings, and checks for binding errors. This can be done without expanding any syncon since they all include their binding semantics. The details of this traversal are tedious and not particularly relevant for this paper and are thus not included, but can be found in the first author's Master's thesis [18]. Two kinds of binding errors are considered:

- An identifier is reported as unbound if it is not part of a binding construct (i.e., if it does not appear in a `#bind` declaration) and is not already bound in its context.
- If a binding for an identifier is introduced twice in the same scope *and* the ranges of the bindings overlap, then they are reported as duplicate. The former requirement allows shadowing, but only in nested scopes, while the latter allows multiple definitions in the same scope, but only if no references could be ambiguous.

Finally, if there are no binding errors, name resolution performs a reference-preserving renaming of all identifiers in the AST, such that no identifier is introduced in a binding more than once. This simplifies writing a correct expansion, since the programmer can now assume that no rearranging of the children of a syncon instance can cause accidental name capture or a duplicate binding.

4.2 Expansion

Expanding a single syncon instance consists of running a simple interpreter for the DSL described in Section 2.3, with one important thing to mention: identifiers that appear in syntax literals must be different between different expansions, to prevent accidental name capture. In practice this is accomplished by tagging each such identifier with a number that is unique per expansion (e.g., the first expansion has number 1, the second number 2, etc.). Note that while the expansion may assume that no rearranging of the instance's children can cause a duplicated binding (as mentioned in the previous section), it does not need to ensure that this is true *after* the expansion.

Expanding an AST repeats this process until no remaining syncon instances have expansions, i.e., until all remaining syncons are part of the base language. However, we do need to maintain the invariant. Since an adjacent binding may affect the AST arbitrarily far away from its introduction (depending on which parents introduce scopes), we cannot simply perform renaming only on the result of the expansion. Instead, we perform another name resolution pass, but only when needed; most syncons do not duplicate children when they expand, in which case there is no possibility of a duplicated binding.

5 Evaluation

The approach presented in this paper has been evaluated through an implementation of the system as a whole, and then creating two language subsets using syncons to evaluate their expressiveness. Note that the language subsets are of general-purpose programming languages, rather than DSLs. Having pre-existing, well-defined semantics gives a ground truth that simplifies evaluation, and the approach being useful for general purpose programming languages suggests that it would be useful for a DSL as well.

5.1 Implementation

To evaluate our approach, we have written an interpreter[1] in Haskell, containing the phases described in sections 2 through 4. The general parser used is an off-the-shelf implementation[2] of the Earley parsing algorithm [7]. We have also implemented a simple interpreter for a small, mostly functional, base language. The language subsets in Section 5.2 expand to this base language, which thus gives us the ability to run programs and compare the syncon language implementations with the original implementations.

The base language features include anonymous functions, mutable references, continuations, and several builtin values (e.g., primitives for arithmetic, list manipulation, and printing).

5.2 Case Studies

To evaluate the expressiveness of syncons, we have implemented small subsets of two common programming languages:

OCaml We have implemented a (dynamically typed) subset of OCaml to test that syncons can express a relatively standard functional programming language, with additional focus on pattern matching. The OCaml subset implementation consists of 32 syncons, spread over 3 syntax types.

Lua We have implemented a subset of Lua to test that syncons can express the control flow common in imperative language. It is worth noting here that tables and coroutines, arguably the more particular features of Lua, are not implemented, since they are not the reason for choosing Lua as a test language. The Lua subset implementation consists of 29 syncons, spread over 5 syntax types.

To test the correctness of these subset implementations, we have written several small programs:

[1] Available at https://github.com/miking-lang/syncon

[2] http://hackage.haskell.org/package/Earley

– `fib.ml` and `fib.lua`. These programs implement functions for finding the *n*th fibonacci number, one with the quadratic recursive definition, and a linear version. They test most binding constructs, some control flow, and basic arithmetic.
– `fizzbuzz.ml` and `fizzbuzz.lua`. These programs implement fizzbuzz, an (in)famous interview problem, and test more control flow and comparisons.
– `misc.ml`. This program tests the various remaining syntax constructions in the OCaml subset, for example, boolean literals, anonymous functions and cons patterns.
– `misc.lua`. This program tests the various remaining syntax constructions in the Lua subset, for example, grouping by parentheses, **`break`**, and multiplication.

The programs are chosen to ensure that between them, each syncon is used at least once, and that some non-trivial control flow is used, e.g., recursion, mutual recursion, and (in Lua) iteration.

We then compare the output of running each program in the subset implementation and the canonical implementation. One additional complexity is that the subset implementations do not support the standard libraries for the languages, nor importing of modules. In particular, printing is a builtin primitive in the subset implementations. To cover this difference we prepend a small prelude to each program before running. With the exception of this prelude, the programs are identical between the implementations.

To test language construct composability, we extend Lua in two ways: we add destructuring to existing binding constructs, and we add a new **`match`**-statement. Both of these are accomplished by reusing syncons from the OCaml subset.

5.3 Analysis and Discussion

This section examines the result of extending the Lua subset with syncons from OCaml, a few ambiguity errors in the OCaml subset, the effects of contextual information on syncon independence, and a brief summary of other results.

Cross-Language Reuse. Fig. 4 shows the example from Section 2, to the left as it was then, and to the right extended to handle destructuring using arbitrary patterns. The patterns are defined in the OCaml subset, but can be reused in Lua by merely importing them. Patterns are implemented as anonymous macros that take two arguments: a function to call if the pattern match fails, and the value to match against. We thus require three changes: switch `Identifier` for `Pattern`, remove **`#bind`** `x` **`after`**, and switch `defAfter` with an invocation of `x`. However, the binding semantics are slightly different, the right version allows `e` to use names introduced by `x`, i.e., we allow recursive bindings, although the system should be extended to be more conservative and disallow such bindings.

To add **`match`** we must accommodate for syntactic and semantic differences between the two languages; sensible syntax in OCaml does not necessarily fit in

```
syntax local:Statement =
  "local" x:Identifier
  ("=" e:Expression)?
{
  #bind x after
  #scope (e)
  BExpression‘
    defAfter ‘id(x) =
    (@ref (@deref
        ‘t(foldr e _ (e)
             (BExpression‘ @unit))))
}
```

```
syntax local:Statement =
  "local" x:Pattern
  ("=" e:Expression)?
{

  #scope (e)
  BExpression‘
    ‘t(x) (fun _. @crash)
    (@ref (@deref
        ‘t(foldr e _ (e)
             (BExpression‘ @unit))))
}
```

Fig. 4. The change required to make a `local`-declaration in Lua support destructuring.

Lua, and Lua has a distinction between statements and expressions while OCaml does not. We thus create a new syncon that expands to the **`match`**-syncon from OCaml, which we import. This choice mirrors one from regular programming: do you use an external library directly, or do you wrap it in an interface that is more convenient for the current application? Different situations will produce different answers.

Ambiguity Errors. Consider the following OCaml code:

```
1  let result = match 1 with
2    | 2 -> match "two" with
3           | str -> str
4    | 4 -> "four"
5  let list = [a; b]
```

Lines 1 through 4 are the example from Section 3.1. Our prototype produces the following ambiguity errors. Since this is a pure research prototype, little effort has been spent on the presentation of the errors, only on what information is presented.

```
Ambiguity: "1:14-4:16"
(("match", "1:14-4:16"),
 [("intLit", "1:20-1:21"),
  ("intPat", "2:5-2:6"),
  ("match", "2:10-3:22"),
  ("intPat", "4:5-4:6"),
  ("stringLit", "4:10-4:16")])
(("match", "1:14-4:16"),
 [("intLit", "1:20-1:21"),
  ("intPat", "2:5-2:6"),
  ("match", "2:10-4:16")])
```

```
Ambiguity: "5:12-5:18"
(("list", "5:12-5:18"),
 [("seqComp", "5:13-5:17")])
(("list", "5:12-5:18"),
 [("variable", "5:13-5:14"),
  ("variable", "5:16-5:17")])
```

The first ambiguity covers the match expression on lines 1 through 4, and has two interpretations. Both are **`match`**-expressions and contain an integer literal (`1`) and an integer pattern (`2`). Then comes the difference: either there is a nested **`match`** ending on line 3, then an integer pattern and a string literal (i.e., another **`match`**-arm), or a single nested **`match`** that ends on line 4. This agrees with the conclusion in Section 3.1: the **`match`**-arm on line 4 could belong to either **`match`**.

The second error states that the right hand side of line 5 is either i) a list of one element (a sequential composition), or ii) a list of two elements (two variables). The rewrite required to handle this case in our prototype as in the canonical implementation is possible, but requires duplicating syncons. An automatic method to perform the rewrite without duplication seems plausible, but is left for future work.

Note that the errors highlight only the ambiguous parts, **`let`** `result =` and **`let`** `list =` are not included, since they are unambiguous.

Contextual Information. Certain syncons require information from their context, e.g., a pattern needs the value being matched, and **`return`** needs to know which function to return from. There are two intuitive ways we might attempt to provide this information:

– Have a parent syncon bind a name which the child uses. This does not work because the system prevents all forms of name capture.
– Have the child syncon produce a function, which the parent syncon then applies to the information required. This works, but the function introduces a scope, which hides any adjacent bindings exposed by the child.

To work around this, our base language contains a form of anonymous macros; functions that do not introduce a scope, must be applied immediately, and take an opaque piece of syntax as an argument.

However, using anonymous macros in this fashion introduces coupling between syncons: if one syncon requires contextual information, then all syncons of the same syntax type must produce an anonymous macro and the information must be threaded to children, even if the syncon itself does not require it.

Other Considerations and Limitations. This work introduces the new syncon approach, which enables both fine-grained composability and syntactic freedom. There are, however, a few limitations with the work presented so far. Specifically, i) the binding semantics for syncons have no concept of modules or namespaces, and ii) syncons cannot be disambiguated by whether their contained identifiers are bound or not, which precludes, e.g., pattern match and unification as done in Prolog. We consider these limitations as future work. Additionally, we wish to evaluate the approach on complete languages, rather than the subsets here, as well as more common language compositions, e.g., HTML and JavaScript.

6 Related Work

Macros in the Lisp family of languages provide a small unit of composition, but are limited in their syntax; a Lisp macro still looks like Lisp. Racket [10], a Lisp language oriented towards language creation, allows a language designer to supply their own parser, circumventing this limitation. However, such a parser loses the small unit of composition; the syntax is defined as a whole rather than as a composition of smaller language constructs. λ_m [13] introduces a type system that prevents macros from introducing binding errors, thus providing an automated expansion checker that our approach currently lacks. However, λ_m only supports nested bindings, adjacent bindings are not expressible. Romeo [22] goes further and allows procedural macros, as opposed to pattern-based ones, and thus features a more powerful macro system than both λ_m and this paper. Binding safety is again ensured by a type system, with what amounts to algebraic data types describing the abstract syntax trees that are processed. However, this system runs into the expression problem; old macros cannot be reused if new constructs are added to the language being transformed. Additionally, both λ_m and Romeo are still constrained to Lisp syntax.

SoundX [16] takes a different approach to macros: they rewrite type derivation trees instead of syntax trees. The resulting macros can be checked to introduce neither binding errors nor type errors. As an additional benefit, they can also use type information present in the derivation but not explicitly present in the source code. However, the concrete syntax (specified with a context free grammar using SDF [12]) of a language described with SoundX has no guarantees on ambiguity, nor a way to deal with any ambiguity that shows up.

Compared to various embedded approaches to DSLs [1,2,4,11,14,24,27], our approach gives greater syntactical flexibility, but less convenient expressive power due to the limited nature of the DSL for specifying the expansion. Wyvern [17] gives a pragmatic alternative to specifying complete new languages; new types can be given custom literal syntax, but the rest of the language is fixed. Silkensen et al. [21] provide an approach for parsing composed grammars efficiently using what amounts to type information for bound identifiers.

JastAdd [8] allows modularly defining languages using an attribute grammar-based system, but requires an external parser. The system allows smaller units of composition than a language, namely modules. However, extra care must be taken to produce features that can be reused with granularity. Silver [25] (a more concise description and example of use can be found in [15]), also based on attribute grammars, includes a parser (Copper [26]) and gives more guarantees under composition: the composed concrete syntax is unambiguous and no attributes are missing. However, syntax is limited to LALR(1) and each extension construct must start with a distinguishing terminal, to signal the transition from core language to extension language.

The work presented in this paper is based on the first author's Master's thesis [18], which has not been formally published before.

7 Conclusion

In this paper, we introduce the concept of a syntactical construct (syncon) that enables both fine-grained composability of language constructs and syntactic freedom for the syntax of the defined language. As a consequence of this flexibility, we show how dynamic ambiguity detection is an alternative to static ambiguity checking. The syncon approach is implemented in Haskell, and evaluated by specifying small subsets of OCaml and Lua using syncons, where certain language constructs are imported from the other language. Although the current implementation has certain limitations, we contend that dynamic ambiguity checking and fine-grained language construct composition can be good complements to pure static and more restrictive approaches. Combining the benefits of both static and dynamic detection—using a form of hybrid detection and reporting—can be an interesting direction for future work.

Acknowledgements The authors would like to thank Elias Castegren, Mads Dam, Philipp Haller, Daniel Lundén, Saranya Natarajan, Oscar Eriksson, and Rodothea Myrsini Tsoupidi for valuable input and feedback. This project is financially supported by the Swedish Foundation for Strategic Research (FFL15-0032).

References

1. Augustsson, L., Mansell, H., Sittampalam, G.: Paradise: a two-stage dsl embedded in haskell. In: Proceedings of the 13th ACM SIGPLAN international conference on Functional programming. pp. 225–228. ICFP '08, ACM (2008)
2. Bjesse, P., Claessen, K., Sheeran, M., Singh, S.: Lava: hardware design in Haskell. In: Proceedings of the third ACM SIGPLAN international conference on Functional programming. pp. 174–184. ACM Press, New York, USA (1998)
3. Bravenboer, M., Kalleberg, K.T., Vermaas, R., Visser, E.: Stratego/XT 0.17. A language and toolset for program transformation. Science of Computer Programming **72**(1), 52–70 (2008)
4. Broman, D., Siek, J.G.: Gradually Typed Symbolic Expressions. In: Proceedings of the ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation. pp. 15–29. PEPM '18, ACM, New York, NY, USA (2018)
5. Cantor, D.G.: On the ambiguity problem of backus systems. J. ACM **9**(4), 477–479 (Oct 1962)
6. Dybvig, R.K., Hieb, R., Bruggeman, C.: Syntactic abstraction in scheme. LISP and Symbolic Computation **5**(4), 295–326 (Dec 1993)
7. Earley, J.: An efficient context-free parsing algorithm. Commun. ACM **13**(2), 94–102 (Feb 1970)
8. Ekman, T., Hedin, G.: The JastAdd system — modular extensible compiler construction. Science of Computer Programming **69**(1), 14 – 26 (2007)
9. Flatt, M.: Binding as sets of scopes. In: Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. pp. 705–717. POPL '16, ACM, New York, NY, USA (2016)

10. Flatt, M., PLT: Reference: Racket. Tech. Rep. PLT-TR-2010-1, PLT Design Inc. (2010)
11. Giorgidze, G., Nilsson, H.: Embedding a functional hybrid modelling language in Haskell. In: Proceedings of the 20th International Symposium on the Implementation and Application of Functional Languages (2008)
12. Heering, J., Hendriks, P.R.H., Klint, P., Rekers, J.: The syntax definition formalism SDF—reference manual—. ACM SIGPLAN Notices **24**(11), 43–75 (nov 1989)
13. Herman, D., Wand, M.: A Theory of Typed Hygienic Macros. Proceedings of the 17th European Symposium on Programming **4960**, 48 (2010)
14. Hudak, P.: Building domain-specific embedded languages. ACM Computing Surveys (1996)
15. Kaminski, T., Kramer, L., Carlson, T., Van Wyk, E.: Reliable and Automatic Composition of Language Extensions to C: The ableC Extensible Language Framework. Proc. ACM Program. Lang. **1**(OOPSLA), 98:1–98:29 (Oct 2017)
16. Lorenzen, F., Erdweg, S.: Sound type-dependent syntactic language extension. In: Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL). vol. 51, pp. 204–216. ACM Press (2016)
17. Omar, C., Kurilova, D., Nistor, L., Chung, B., Potanin, A., Aldrich, J.: Safely Composable Type-Specific Languages. In: Jones, R. (ed.) ECOOP 2014 – Object-Oriented Programming. pp. 105–130. Lecture Notes in Computer Science, Springer Berlin Heidelberg (2014)
18. Palmkvist, V.: Building Programming Languages, Construction by Construction. Master's thesis, KTH Royal Institute of Technology (2018)
19. Rompf, T., Odersky, M.: Lightweight Modular Staging: A Pragmatic Approach to Runtime Code Generation and Compiled DSLs. In: Proceedings of the Ninth International Conference on Generative Programming and Component Engineering. pp. 127–136. GPCE '10, ACM, New York, NY, USA (2010)
20. Sheard, T., Jones, S.P.: Template Meta-programming for Haskell. In: Haskell '02: Proceedings of the 2002 ACM SIGPLAN workshop on Haskell. pp. 1–16. ACM Press, New York, USA (2002)
21. Silkensen, E., Siek, J.: Well-Typed Islands Parse Faster. Springer (2013)
22. Stansifer, P., Wand, M.: Romeo. In: Proceedings of the 19th ACM SIGPLAN international conference on Functional programming - ICFP '14. vol. 49, pp. 53–65. ACM Press, New York, New York, USA (2014)
23. Steele, Jr., G.L.: An overview of COMMON LISP. In: Proceedings of the 1982 ACM symposium on LISP and functional programming. pp. 98–107. LFP '82, ACM, New York, NY, USA (1982)
24. Sujeeth, A.K., Brown, K.J., Lee, H., Rompf, T., Chafi, H., Odersky, M., Olukotun, K.: Delite: A Compiler Architecture for Performance-Oriented Embedded Domain-Specific Languages. ACM Transactions on Embedded Computing Systems (TECS) **13**(4s), 134:1–134:25 (Apr 2014)
25. Van Wyk, E., Bodin, D., Gao, J., Krishnan, L.: Silver: An extensible attribute grammar system. Science of Computer Programming **75**(1-2), 39–54 (jan 2010)
26. Van Wyk, E.R., Schwerdfeger, A.C.: Context-aware scanning for parsing extensible languages. In: Proceedings of the 6th international conference on Generative programming and component engineering - GPCE '07. p. 63. ACM Press, New York, New York, USA (2007)
27. Wan, Z., Hudak, P.: Functional reactive programming from first principles. In: PLDI '00: Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation. pp. 242–252. ACM Press, New York, USA (2000)