Research Report: sslang

Leo Qiao (flq2101)

May 13, 2022

1 Introduction

This research report gives an overview of my research progress during 2022 Spring as a member of the Type System team within the sslang research group. The team is headed by John Hui and consists of Xijiao Li and Leo Qiao (me). This report discusses my work on patterns and typeclasses.

2 Patterns

2.1 Problems

Patterns in sslang AST is much more expressive than patterns in sslang IR. As a result, prior to work in this semester, sslang does not support several features for patterns matching, including recursive patterns and at-patterns. In addition, the patterns are not checked for anomalies, such as inexhaustion and useless clauses.

The first problem is solved by adding an AST pass that desugars complex pattern matches to simpler pattern matches that can be properly lowered into IR, and the second problem is solved by adding an anomaly check before the desugar pass that checks whether the patterns are well formed, exhaustive and free of useless clauses.

It should be noted that the desugar pass is general enough that it can be trivially applied to function defined with pattern arguments, because the algorithm is designed to handle multiple parallel pattern-matches instead of just a single pattern-match. This can be added in the future when it is more clear what function defined with pattern arguments would look like in sslang.

2.2 Anomaly Checks

The sslang pattern-match anomaly checks reference *Warnings for pattern matching* by Luc Maranget.

Note that the anomaly checks here are specialized for strict languages (such as sslang). For adaptations for lazy languages, refer to the original paper.

In addition, this section does not walk through the proof of correctness for the algorithms. It solely focuses on the implementation.

I first provide a simple, generalized set of definitions that would ease the formalization of the algorithms. Next, the anomaly check algorithms are discussed in the context of the generalized definitions. Finally, I discuss how the generalized definitions and algorithms are adapted for the sslang AST.

2.2.1 Generalized Definitions

For simplicity, let's define values to be data constructor terms:

v ::=		**(Defined) values**
	$\mathrm{c}(\mathrm{v}_1, v_2, ..., v_a)$	$\mathrm{a} \geq 0$

Patterns are used to describe a set of values with a common prefix. An additional wildcard pattern is used to describe the set of all values.

p ::=		**Patterns**
	_	wildcard
	$\mathrm{c}(\mathrm{p}_1, p_2, ..., p_a)$	constructed pattern, $\mathrm{a} \geq 0$

We use $\vec{p} = (p_1 ... p_a)$ to denote a vector of patterns and $\vec{v} = (v_1 ... v_a)$ to denote a vector of values.

We will also consider matrices of patterns $P = (p^i_j)$ of size $m \times n$, where m is the number of rows and n is the number of columns. Let $\emptyset$ denote matrices of size with no row ($m = 0$ and $n \geq 0$), and let $()$ denote non-empty matrices of empty rows ($m > 0$ and $n = 0$). The reason for considering matrices of patterns should become clearer in the next subsection.

2.2.2 Usefulness check

It turns out that the identification of both inexhaustion and useless clause can be easily achieved by using a so-called *usefulness* check. The usefulness check is a pure function $\mathcal{U}(P, \vec{q})$, which takes a matrix of size $m \times n$ and a vector of size n as input and returns a boolean as output. Informally, the boolean indicates whether q would be useful if appended to P.

In addition, it should be noted that the algorithms here assumes well-typed programs. This is a property that does not hold for sslang AST. In the next section, I will discuss how this is accounted for and resolved.

2.2.3 Behavior and usage of $\mathcal{U}(P, \vec{q})$

To keep the report clear and succinct, I want to avoid too much abstract formalization, so the meaning and usage of the usefulness check $\mathcal{U}$ is illustrated with examples.

Suppose we have the following program.

```
type Bool
  True
  False

type List
  Cons Bool List
  Nil

any l = match l
  Cons True  l' = True
  Cons False l' = any l'
  Nil           = False
  Cons _     _  = False
```

First, let's check for useless clauses in the pattern matching.

We would transform the patterns into a matrix of size $m = 4$ and $n = 1$:

$$\begin{pmatrix} Cons(True, \_) \\ Cons(False, \_) \\ Nil() \\ Cons(\_, \_) \end{pmatrix}$$

To check for useless clauses, we would perform a series of usefulness checks. The first check is

$$\mathcal{U}(P, \vec{q}) = \mathcal{U}(\emptyset, (Cons(True, \_)))$$

, which checks whether the first pattern is useful. If this is true, then we proceed to the second check

$$\mathcal{U}(P, \vec{q}) = \mathcal{U}(\big(Cons(True, \_)\big), (Cons(False, \_)))$$

, which checks whether the second pattern is useful. The same process repeats for each row in the original matrix, and the first row for which the usefulness check returns false would be a useless clause. In this case, the following check returns false:

$$\mathcal{U}(P, \vec{q}) = \mathcal{U}(\begin{pmatrix} Cons(True, \_) \\ Cons(False, \_) \\ Nil() \end{pmatrix}, (Cons(\_, \_)))$$

. This means that the fourth pattern is useless.

Next, let's perform exhaustion check on the same program.

Using the same matrix as above, the exhaustion check problem reduces to the following usefulness check:

$$\mathcal{U}(P, \vec{q}) = \mathcal{U}(\begin{pmatrix} Cons(True, \_) \\ Cons(False, \_) \\ Nil() \\ Cons(\_, \_) \end{pmatrix}, (\_))$$

. Intuitively, if adding a wildcard pattern is useful, then the original sequence of patterns is inexhaustive; otherwise, the sequence is exhaustive.

2.2.4 Recursive algorithm for $\mathcal{U}$

It is clear now that an efficient algorithm for $\mathcal{U}(P, \vec{q})$ would unlock the anomaly checks for us. This section describes a recursive algorithmic function for $\mathcal{U}(P, \vec{q})$. I will overload the name $\mathcal{U}$ to refer to the algorithmic function here as well.

Let P be a pattern matrix of size $m \times n$ and $\vec{q}$ be a pattern vector of size n. The inductive process proceeds by decomposing P and $\vec{q}$ along the first column.

Base Case: $n = 0$**.**

The base cases are:

$$\mathcal{U}((), ()) = False$$

$$\mathcal{U}(\emptyset, \vec{q}) = True$$

. Intuitively, the second case is saying that adding any pattern to a matrix with no rows is useful. As for the first case, it is more of a technicality for the recursion to work.

Inductive Case: $n > 0$**.**

There are two sub-cases depending on the pattern p_1.

1. Pattern q_1 is a constructed pattern, that is $q_1 = c(r_1, ..., r_a)$. From matrix P, we calculate a *specialized* matrix $\mathcal{S}(c, P)$. The new matrix $\mathcal{S}(c, P)$ has $a+n-1$ columns and its rows are defined from the rows of P, according to the first component of these rows using the following rules:

p_1^i	$\mathcal{S}(c, P)$
$c(r_1, ..., r_a)$	$r_1 \ ... \ r_a \ p_2^i \ ... \ p_n^i$
$c'(r_1, ..., r_{a'})$	No Row
_	$\_ \ ... \ \_ \ p_2^i \ ... \ p_n^i$

   Let's also define the behavior of $\mathcal{S}(c, \vec{q})$:

$$\mathcal{S}(c, (c(r_1, ..., r_a) \ q_2 \ ... \ q_n) = (r_1 \ ... \ r_a \ q_2 \ ... \ q_n)$$

$$\mathcal{S}(c, (\_ \ q_2 \ ... \ q_n) = (\_ \ ... \ \_ \ q_2 \ ... \ q_n)$$

   (note: that returned value in the wildcard case has a wildcards appended to the rest of the vector).

   With these definitions of $\mathcal{S}$, the definition of $\mathcal{U}$ in this sub-case is:

$$\mathcal{U}(P, \vec{q}) = \mathcal{U}(\mathcal{S}(c, P), \mathcal{S}(c, \vec{q}))$$

   .

   Intuitively, the specialized matrices are specializing P to the pattern vectors that start with the constructor in question.

2. Pattern q_1 is a wildcard, that is $q_1 = \_$. Let $\Sigma = c_1, c_2, ..., c_z$ be the set of constructors that appear as root constructors of the patterns of P's first column.

   If Σ **is a complete signature**, we define

$$\mathcal{U}(P, \vec{q}) = \bigvee_{k=1}^{z} \mathcal{U}(\mathcal{S}(c_k, P), \mathcal{S}(c_k, \vec{q}))$$

   If Σ **is not a complete signature**, we calculate a *default* matrix $\mathcal{D}(P)$ of width $n-1$ and its rows are defined using the following rules:

p_1^i	$\mathcal{D}(P)$
$c_k(r_1, ..., r_a)$	No Row
_	$p_2^i \ ... \ p_n^i$

We can then define

$$\mathcal{U}(P, (\_ \ q_2 \ ... \ q_n)) = \mathcal{U}(\mathcal{D}(P), (q_2 \ ... \ q_n))$$

Intuitively, this means that we will only need to examine the wildcard cases.

This concludes the definition of the algorithm $\mathcal{U}$. We now have a well-defined anomaly check algorithm.

2.2.5 Adapting to sslang AST

The sslang AST has the following pattern definition:

```
data Pat
  = PatWildcard            -- ^ Match anything, i.e., @_@
  | PatId Identifier       -- ^ Variable or data constructor, e.g., @v@ or @Some@
  | PatLit Literal         -- ^ Literal match, e.g., @1@
  | PatAs Identifier Pat   -- ^ Pattern alias, e.g., @a \@ <pat>@
  | PatTup [Pat]           -- ^ Match on a tuple, e.g., @(<pat>, <pat>)@
  | PatApp [Pat]           -- ^ Match on multiple patterns, e.g., @Some a@
  | PatAnn Typ Pat         -- ^ Match with type annotation, e.g., @<pat>: Type@
```

It is clear that there are a lot more patterns than in the generalized pattern definition, but the original algorithm can still be applied with some extensions. It is easy to see that variable constructors behaves exactly the same as wildcards. In theory, PatTup and PatLit behave similarly to constructed patterns, so their differences can be accounted for through practical software engineering. PatLit requires special attention during implementation, as determining whether Σ is a complete signature is a little involved in this case.

To account for PatAs and PatAnn, the specialized matrix $\mathcal{S}(P)$ and default matrix $\mathcal{D}(P)$ can be extended trivially. The extension to specialized matrix is given by

p_1^i	$\mathcal{S}(c, P)$
$x@r$	$\mathcal{S}(c, \begin{pmatrix} r & p_w^i \ ... \ p_n^i \end{pmatrix})$
$r : t$	$\mathcal{S}(c, \begin{pmatrix} r & p_w^i \ ... \ p_n^i \end{pmatrix})$

Plainly, the additional things are removed. Similarly, the extension to default matrix is given by

p_1^i	$\mathcal{D}(P)$
$x@r$	$\mathcal{D}(\begin{pmatrix} r & p_w^i \ ... \ p_n^i \end{pmatrix})$
$r : t$	$\mathcal{D}(\begin{pmatrix} r & p_w^i \ ... \ p_n^i \end{pmatrix})$

.

2.2.6 Well-formed check

Since sslang AST has not been type-checked (or even type inferred), some well-formed checks are run prior to each call of $\mathcal{U}$. They check that the root constructors are of the same type and have the correct number of arguments supplied to them. Here, the root constructors include tuples and literals.

2.3 Desugar Patterns

The sslang pattern match desugaring pass references *The Implementation of Functional Programming Languages* by Simon L. Peyton Jones. In particular, it references Chapter 5 *Efficient Compilation of Pattern-Matching* by Philip Wadler.

Note that the original algorithm assumes well-typed programs. This assumption is partially (but not completely) satisfied by the well-formness checks and the anomaly checks described in the previous section.

The formalization of the algorithm described in this section differs from the original book, because the book uses specific theoretical languages (e.g. Miranda) to discuss the algorithm. Here, I try to discuss the algorithm without references to the theoretical languages.

I first provide a simple, generalized set of definitions. Next, the desugar algorithm is discussed in the context of the generalized definitions. Finally, I discuss how the generalized definitions and algorithms are adapated for the sslang AST.

2.3.1 Generalized Definitions

For simplicity, let's define a pattern to be either a variable or a constructor pattern:

```
type Variable = String
type Constructor = String

data Pat
  = PatVar Variable       -- ^ Match anything, e.g., @x@
  | PatCon Constructor [Pat] -- ^ Constructor pattern, e.g., @Cons x xs@
```

Notice that PatCon can be a recursive pattern.

The following is a program that we will try to desugar.

```
type Bool
  True
  False

type List
  Cons Bool List
  Nil

type Tuple
  Tuple3 List List List

demo x =
  match x
    Tuple3 f Nil          ys          = A f ys
    Tuple3 f (Cons x xs) Nil          = B f x xs
    Tuple3 f (Cons x xs) (Cons y ys) = C f x xs y ys
```

Note that this program also illustrates how we could try to desugar functions defined with pattern arguments.

2.3.2 Algorithm

The following function will be the desugar algorithm:

```
type Equation = ([Pat], Expr)

desugar :: [Variable] -> [Equation] -> Expr -> Expr
desugar us qs def = undefined
```

The pattern-match in demo will be desugared with this function call:

```
desugar [x]
        [ ([Tuple3 f Nil          ys          ], A f ys       )
        , ([Tuple3 f (Cons x xs) Nil         ], B f x xs     )
        , ([Tuple3 f (Cons x xs) (Cons y ys)], C f x xs y ys)
        ]
        catchAllError
```

The def in the function definition of desugar is a default expression. It will be clear why this is necessary. For the initial call, the catchAllError value is passed in. This value represents some expression that

acts as a catch-all error function that would be returned if the pattern-match is inexhaustive and x is not a member of any of the patterns. Since we have inexhaustion check, this catchAllError will never be used and it is thus immaterial what it really is for sslang specifically.

The definition of desugar function can be described as a combination of four distinct rules, namely: variable rule, constructor rule, empty rule and mixture rule. We will discuss them one by one in an abstract manner, and then apply the rules to desugar the demo pattern-match.

2.3.3 The variable rule

This rule applies when every equation begins with a variable pattern. This means that the call of desugar will have the form:

```
desugar (u : us)
        [ ( (v_1 : ps_1), e_1 )
        , ...
        , ( (v_m : ps_m), e_m )
        ]
        e
```

This should be reduced to

```
desugar (u : us)
        [ ( ps_1, subst u v_1 e_1 )
        , ...
        , ( ps_m, subst u v_m e_m )
        ]
        e
```

where subst u v e means "substitute occurrences of variable u in e with variable v".

2.3.4 The constructor rule

This rule applies when every equation begins with a constructor pattern. This is the most complex rule and deserves an example walk-through prior to the abstract definition.

Suppose we have the following call to desugar:

```
desguar [u1]
        [ ( [Cons x Nil         ], B x       )
        , ( [Nil                ], A         )
        , ( [Cons y (Cons x xs)], C y x xs )
        ]
        e
```

First, group the equations that begin with the same constructor. Notice that this is always safe.

```
desguar [u1]
        [ ( [Nil                ], A         )
        , ( [Cons x Nil         ], B x       )
        , ( [Cons y (Cons x xs)], C y x xs )
        ]
        e
```

Next, we can reduce this to the following expression:

```
match u1
  Nil        = desguar []
                       [ ( [], A ) ]
                       e
  Cons u2 u3 = desguar [u2, u3]
                       [ ( [x, Nil      ], B x       )
                       , ( [y, Cons x xs], C y x xs )
                       ]
                       e
```

Notice that Nil introduces no new variables (leaving a call of desugar with an empty list of variables), and Cons introduces two new variables, u_2 and u_3.

As a side-note, suppose that we don't have the exhaustion check and we have the following inexhaustive call to desugar:

```
desugar [u1]
        [ ( [Cons x Nil         ], B x       )
        , ( [Cons y (Cons x xs)], C y x xs )
        ]
        e
```

This would get reduced to

```
match u1
  Nil         = desugar [] [] [] e
  Cons u2 u3 = desugar [u2, u3]
                       [ ( [x, Nil       ], B x      )
                       , ( [y, Cons x xs], C y x xs)
                       ]
                       e
```

Plainly, the desugared pattern-match should still contain a clause for the missing constructor.

Hopefully, the examples above helped with intuition, so we can now describe the rule abstractly. Suppose that the constructors are from a type which has constructors c_1, ..., c_k. Then the equations can be rearranged into groups of equations qs_1, ..., qs_k, such that every equation in group qs_i begins with constructor c_i. The call to desugar will then have the form:

```
desugar (u : us) (qs1 ++ .. ++ qsk) e
```

where each qs_i has the form:

```
[ ( ((ci ps'i,1) : psi,1), ei,1 )
, ...
, ( ((ci ps'i,ai) : psi,ai), ei,ai )
]
```

and a_i denotes the arity of c_i. The constructor pattern (c p_1 ... p_k) has also been abbreviated to the form (c ps). This call to desugar is reduced to:

```
match u
  c1 us'1 = desugar (us'1 ++ us) qs'1 e
  ...
  ck us'k = desugar (us'k ++ us) qs'k e
```

where each qs'$_i$ has the form

```
[ ( (ps'i,1 ++ psi,1), ei,1 )
, ...
, ( (ps'i,ai ++ psi,ai), ei,ai )
]
```

and each us'$_i$ is a list of new variables, containing one variable for each field of c_i.

2.3.5 The empty rule

This rule applies when we arrive at a call of desugar where the variable list is empty, such as

```
desugar []
        [ ( [], A u1 u3 ) ]
        e
```

This reduces to

```
A u1 u3
```

.

In general, the list of equations may have zero, one or more equations. Thus, the general form of a call of desugar with an empty variable list is

```
desugar []
        [ ( [], e_1 )
        , ...
        , ( [], e_m )
        ]
        e
```

where $m \geq 0$. This can be reduced to e_1 if $m > 0$ and e if $m = 0$.

2.3.6 The mixture rule

This rule applies when there is a mixture of equations that begin with a variable and equations that begin with a constructor. This means that calls of desugar would look like the following:

```
desugar us qs e
```

where qs may be partitioned into k lists qs_1, ..., qs_k such that

```
qs = qs_1 ++ ... ++ qs_k
```

. The partition is chosen so that each qs_i either has every equation beginning with a variable or every equation beginning with a constructor. Then, the call of desugar can be reduced to:

```
desugar us qs_1 (desugar us qs_2 ( ... (desugar us qs_k e)...))
```

Here, we are making use of the default expression parameter to chain the desugared sub-pattern-matches together.

2.3.7 Example walk-through: demo function

Let us go through the process with the demo function.

Desugaring the demo function becomes the following call to desugar:

```
desugar [x]
        [ ([Tuple3 f Nil          ys          ], A f ys       )
        , ([Tuple3 f (Cons x xs) Nil         ], B f x xs     )
        , ([Tuple3 f (Cons x xs) (Cons y ys)], C f x xs y ys)
        ]
        catchAllError
```

Apply the constructor rule and reduce to:

```
match x
  Tuple3 u_1 u_2 u_3 =
    desugar [u_1, u_2, u_3]
            [ ([f, Nil         , ys          ], A f ys )
            , ([f, (Cons x xs), Nil         ], B f x xs )
            , ([f, (Cons x xs), (Cons y ys)], C f x xs y ys )
            ]
            catchAllError
```

Apply the variable rule and reduce to:

```
match x
  Tuple3 u_1 u_2 u_3 =
    desugar [u_2, u_3]
            [ ([Nil         , ys          ], A u_1 ys )
            , ([(Cons x xs), Nil         ], B u_1 x xs )
            , ([(Cons x xs), (Cons y ys)], C u_1 x xs y ys )
            ]
            catchAllError
```

Apply the constructor rule and reduce to:

```
match x
  Tuple3 u1 u2 u3 =
    match u2
      Nil =
        desugar [u3]
                [ ([ys], A u1 ys ) ]
                catchAllError
      Cons u4 u5 =
        desugar [u4, u5, u3]
                [ ([x, xs, Nil         ], B u1 x xs )
                , ([x, xs, (Cons y ys)], C u1 x xs y ys )
                ]
                catchAllError
```

Apply the variable rule enough times and reduce to:

```
match x
  Tuple3 u1 u2 u3 =
    match u2
      Nil =
        desugar []
                [ ([], A u1 u3 ) ]
                catchAllError
      Cons u4 u5 =
        desugar [u3]
                [ ([Nil         ], B u1 u4 u5 )
                , ([(Cons y ys)], C u1 u4 u5 y ys )
                ]
                catchAllError
```

Apply empty rule for the top one and constructor rule for the bottom one:

```
match x
  Tuple3 u1 u2 u3 =
    match u2
      Nil = A u1 u3
      Cons u4 u5 =
        match u3
          Nil =
            desugar []
                [ ([], B u1 u4 u5 ) ]
                catchAllError
          Cons u6 u7 =
            desugar [u6, u7]
                [ ([y, ys], C u1 u4 u5 y ys ) ]
                catchAllError
```

Finally, apply the empty for the top one and variable rule multiple times for the bottom one and we get:

```
match x
  Tuple3 u1 u2 u3 =
    match u2
      Nil = A u1 u3
      Cons u4 u5 =
        match u3
          Nil = B u1 u4 u5
          Cons u6 u7 = C u1 u4 u5 u6 u7
```

2.3.8 Adapting to sslang AST

Again, here is the pattern definition in sslang AST:

```
data Pat
  = PatWildcard           -- ^ Match anything, i.e., @_@
  | PatId Identifier      -- ^ Variable or data constructor, e.g., @v@ or @Some@
  | PatLit Literal        -- ^ Literal match, e.g., @1@
  | PatAs Identifier Pat  -- ^ Pattern alias, e.g., @a \@ <pat>@
  | PatTup [Pat]          -- ^ Match on a tuple, e.g., @(<pat>, <pat>)@
  | PatApp [Pat]          -- ^ Match on multiple patterns, e.g., @Some a@
  | PatAnn Typ Pat        -- ^ Match with type annotation, e.g., @<pat>: Type@
```

We can extend the original algorithm to desugar these additional patterns. PatWildcard behaves similarly to the variable patterns; the only difference is that no substitution is required. In theory, PatTup and PatLit behave similarly to constructor patterns, so their differences can be accounted for through practical software engineering. Desugaring as-patterns reduces to similar behavior as substitution in variable rule. Desugaring annotation-pattern is achieved by inserting a type-annotated let-binding in the match-clause expression body.

Since it is still unclear what the substitution scheme is for sslang, inserting let-bindings is used instead of actual variable substitution.

Since sslang pattern-matches admits an expression after the match keyword, if the expression is not just a variable, the expression is first found to a fresh variable in a let-binding, and the fresh variable is instead used for the matching. This is done to be able to call the desugar function on the pattern-match.

3 Typeclass

The work on typeclass this semester has been purely theoretical and exploratory. For this report, since I don't have enough time to fully discuss the various concepts that I have encountered, I will leave details to a future report and propose some goals to pursue over the summer.

From the work in previous semester, we already know how to use dictionary-passing technique to desugar type classes and instances into ordinary ADT's. A simple implementation was written in the typeclass-instantiation branch.

The most important and difficult challenge now is how to alter our type system to support typeclass. For this feature, we have several things to consider, including:

- Do we want higher-kinded typeclasses?
- How do we want to type-check/type-infer typeclasses?

To explore the possibility of supporting higher-kinded typeclasses, there are a few very useful papers; in particular, we should refer to *A system of constructor classes: overloading and implicit higher-order polymorphism* by Mark P. Jones. The paper introduces a type system that extends Hindley-Milner to add support for typeclasses; the concept of qualified types is introduced in addition to the types and type schemes in HM. A kind system and a kind-inference algorithm are provided as well. The details of these concepts will be left for a future report (possibly by the end of summer); it deserves to be discussed in a lengthier report with much more context.

The interesting thing is that the qualified types look like constraints. We have been studying the constraint-based HM type system as a group together, and it looks like this approach could potentially solve several of our problems, including typeclass constraints. It turns out that Haskell GHC does indeed do type inference as constraint solving, so a goal would be to study how such systems work and possibly implement a smaller, prototype version of it for sslang. In the end, we can compare this approach with the one introduced in Jones's paper.

For these and many other things to happen, I agree with John that sslang deserves a rewrite. In particular, I support the idea of having a richer, slightly more expressive IR. Now that we have more experience in type systems, we can make our type system constructs easier to use. A more expressive IR would also allow us to first lower the AST to IR and do type-inference first before every other pass, which is a much more intuitive and sane way of satisfying "well-typed" assumptions of many other passes.