Abstract of "An Inside-Out Resugaring System" by Sorawee Porncharoenwase, Brown University, May 2018.

Desugaring is a process that transforms a program with syntactic sugar into a program without it. The process is widely employed in many programming languages. However, desugaring loses information, making it interact poorly with some tools, in particular algebraic steppers. Resugaring is a process that recovers this lost information, so that tools like algebraic steppers can work correctly. However, the previous work on resugaring required an outside-in desugaring order, which has some limitations. In this thesis, we demonstrated that it is possible to create a resugaring system with an inside-out desugaring order, which overcomes these limitations.

An Inside-Out Resugaring System

by
Sorawee Porncharoenwase

A Thesis submitted in partial fulfillment of the requirements for Honors
in the Department of Computer Science at Brown University

Providence, Rhode Island
May 2018

This thesis by Sorawee Porncharoenwase is accepted in its present form by the Department of Computer Science as satisfying the research requirement for the awardment of Honors.

Date _____ _____

Shriram Krishnamurthi, Reader

Date _____ _____

Justin Pombrio, Reader

Acknowledgements

I would like to express my gratitude to Shriram Krishnamurthi, Justin Pombrio, and Tim Nelson for their guidance for the past four years. They are not only excellent programming language teachers, but also great English teachers who taught me how to reason and write. I would not have been able to finish this thesis without their support.

I also would like to thank Jittat Fakcharoenphol who has been my role model that led me to become interested in computer science.

I would like to give thanks to all of my friends, including the people in programming languages labs, Brown's Thai Student Association, and the Brown CS community. In addition, I would like to thank Pakawut Jiradilok, Abhabongse Janthong, and Pakapol Supaniratisai who have helped me throughout my undergraduate studies. I thank Jasper Lee and Clayton Sanford for their feedback about my thesis presentation.

Lastly, I would like to thank my family for all their love and their encouragement.

Contents

1 Introduction and Background **1**

1.1 Resugaring . . . 2

1.2 Pattern-based Desugaring . . . 3

1.3 Our Contribution . . . 4

2 A Desugaring System for Pyret **6**

2.1 Terms . . . 6

2.2 Desugaring . . . 7

2.3 Patterns . . . 10

2.3.1 Primitives . . . 10

2.3.2 Pattern variables . . . 10

2.3.3 Surface and Core . . . 10

2.3.4 Auxiliaries . . . 11

2.3.5 Wildcards . . . 11

2.3.6 Lists (without ellipses) . . . 11

2.3.7 Lists (with ellipses) . . . 11

2.3.8 Fresh, Capture, and Variables . . . 12

2.3.9 Metafunctions and Bijections . . . 12

3 Internals **14**

3.1 Overview . . . 14

3.2 Matching, Substitution, and Specialization . . . 15

3.2.1 Tag Terms . . . 16

3.2.2 Tracking User's Code . . . 16

3.2.3 Environment . . . 16

3.2.4 Matching . . . 16

3.2.5 Substitution . . . 18

3.3 Algebraic Stepper . . . 18

4 Conclusion **20**
4.1 Evaluation 20
4.2 Ongoing Work 20
4.3 Future Work 21

Bibliography **22**

A Sugar in Pyret **23**

Chapter 1

Introduction and Background

In programming languages, syntax can be categorized into two groups: *syntactic sugar* and *core syntax*. Syntactic sugar (or simply *sugar*[1]) is a shorthand for a combination of core syntax[2] [5]. It lets users write programs more concisely and perhaps more aesthetically pleasingly. Core syntax, in contrast, is simply ordinary syntax that is not sugar. For example, the array access syntax `α[ι]` in the C language is sugar that stands for a combination of pointer dereference and addition `*(α + ι)` [2]. The or syntax `(or α β)` in the Racket language is sugar that stands for a combination of let and if `(let ([x α]) (if x x β))`[3]. Hence, the statements in Fig. 1.1a and Fig. 1.1b are semantically equivalent, and so are the expressions in Fig. 1.1c and Fig. 1.1d.

For the reasons mentioned above, sugar is useful for language users[4]. However, it is a burden for language developers who create tools (e.g., interpreters, compilers, and type checkers) because in addition to core syntax, they need to take sugar into consideration. Moreover, they could make a

[1] Sugar is also known as *macro* when it is user-defined.

[2] In practice, we relax this definition so that sugar can be a shorthand for a combination of any syntax, provided that it is *transitively* a shorthand for a combination of core syntax.

[3] To be accurate, `x` must be *fresh*. See more details at Section 2.3.8.

[4] There are criticisms on how too much sugar could be overwhelming to users, however, although this is out of scope of our work.

```c
int arr = {6, 2, 8, 3, 1, 8};
printf("%d\n", arr[5]);
```

(a) A snippet of code in C which contains an array access sugar

```c
int arr = {6, 2, 8, 3, 1, 8};
printf("%d\n", *(arr + 5));
```

(b) A corresponding snippet where the array access sugar is expanded to what it stands for

```racket
(not (or (not #t) (not #f)))
```

(c) A snippet of code in Racket which contains an or sugar

```racket
(not (let ([x (not #t)])
       (if x x (not #f))))
```

(d) A corresponding snippet where the or sugar is expanded to what it stands for

Figure 1.1: Examples of sugar in C and Racket

mistake in the implementation of sugar, making the semantics of sugar mismatch the semantics of what it is meant to stand for.

Often, language developers use a process called *desugaring* to help decrease the burden. Desugaring is a program transformation that turns a program with sugar into a program without sugar by simply *expanding* it into what it stands for until none is left. It typically operates on the *abstract syntax tree* (AST), an internal representation of a program after the program has been parsed. With desugaring, language developers can simply process the desugared program. In a sense, desugaring lets them pretend that sugar doesn't exist. Moreover, because the semantics of sugar is defined by the semantics of what it stands for, desugaring guarantees by construction that sugar will have the correct semantics.

The term *core language* is used to describe the language that has core syntax but doesn't have any sugar. In contrast, the term *surface language* is used to describe the original language. Desugaring is thus a transformation from a surface language to its corresponding core language.

1.1 Resugaring

Unfortunately, the desugaring process loses information. In particular, two different programs in the surface language could desugar to the same program in the core language. Thus, by looking at the desugared program alone, we would not always be able to tell what the original program looks like. In many applications where only semantics of a program matters, like interpretation, this is totally fine. For applications that involves syntax, however, this could be undesirable. One example is *algebraic stepper* [3] which takes a program as an input, and displays evaluation steps until the evaluation finishes, or continues forever if the input program is stuck in an infinite loop. See Fig. 1.2 for an example. If language developers first desugar the program and then naïvely run the stepper, the stepping will be done on the desugared program, which is in the core language. This is unacceptable because users did not write the desugared program, so it would be wrong to show its evaluation steps (compare Fig. 1.2a and Fig. 1.2b). Hence, language developers need to explicitly handle every sugar, thereby losing the benefits of desugaring.

Pombrio et al. proposed a *resugaring* framework for evaluation steps[5], which lifts evaluation steps in the core language up to the surface language so that users can see evaluation steps in terms of what they wrote [6]. This allows language developers who want to support an algebraic stepper to employ desugaring again, restoring the benefits of desugaring. The key idea is to keep enough information from the original program so that the desired evaluation steps can be reconstructed from the evaluation steps of the desugared program. Nonetheless, keeping information manually is very tedious and prone to errors. Pombrio et al. thus created a domain-specific language that lets language developers write *pattern-based desugaring rules* (described in Section 1.2). In the desugaring phase, in addition to expanding sugar, their resugaring framework also tags information from

[5]The term 'resugaring' has subsequently been used for type rules and scope rules as well, but in the context of our work, we use it exclusively for evaluation step resugaring.

```
   (not (or (not #t) (not #f)))
-> (not (or #f (not #f)))
-> (not (not #f))
-> (not #t)
-> #f
```

(a) An output from an algebraic stepper where the input is `(not (or (not #t) (not #f)))`.

```
   (not (let ([x (not #t)])
          (if x x (not #f))))
-> (not (let ([x #f])
          (if x x (not #f))))
-> (not (if #f #f (not #f)))
-> (not (not #f))
-> (not #t)
-> #f
```

(b) An output from an algebraic stepper where the input is the desugared program of Fig. 1.2a and `or` is the only sugar.

Figure 1.2: Examples of algebraic stepper output

the original program. This recovers the lost information during desugaring, and makes evaluation step resugaring possible.

1.2 Pattern-based Desugaring

In a pattern-based desugaring language, there are several desugaring rules and each rule has left-hand side (LHS) pattern and right-hand side (RHS) pattern. A LHS pattern defines a sugar form, and its corresponding RHS pattern defines what it stands for.

A concrete AST node is known as a *term*, and it can be represented using S-expression. For example, the expression `arr[i + 1][j - 1]` in C can be represented as a term as:

```
(array-access (array-access (id arr)
                            (plus (id i) (num 1)))
              (minus (id j) (num 1)))
```

Given a term, its *expansion* proceeds by finding the first desugaring rule whose LHS matches, and it will be expanded to another term using the structure of the RHS pattern. To desugar an entire term, all sugar terms in the said term are expanded. For example, with the rule:

```
(array-access α ι) => (dereference (plus α ι))
```

the above term desugars to:

```
(dereference (plus (dereference (plus (id arr)
                                      (plus (id i) (num 1))))
                   (minus (id j) (num 1))))
```

The order of expansion doesn't matter in the above example, but some desugaring rules are sensitive to the order of expansion. The framework by Pombrio et al. expands sugar in the *outside-in* order, meaning that the outermost sugar is expanded first. The outside-in strategy is very expressive. However, it has three disadvantages:

1. The outside-in order does not allow *helper sugar*. In programming, it is typical to write helper functions to abstract out common functionality in functions, or to break up a long function

into several components. Helper sugar serves the same purpose in desugaring. Unfortunately, with the outside-in expansion order, the inner sugar terms will not have been expanded yet when the outermost sugar term is being expanded. For example, with a desugaring rule `(A <x>) => (B (helper <x>))` and a term `(A 0)`, after `A` is expanded, we will have a term `(B (helper 0))`. Here, `B` will be unable to match the result of the expansion of `(helper 0)`; instead it will match against `(helper 0)`, which is undesirable. While it is possible to rewrite desugaring rules to avoid this problem, the fact that the outside-in order does not support this common abstraction is a disadvantage[6].

2. It is not clear how to statically check that the desugaring rules will preserve the well-formedness of the abstract syntax tree with the outside-in order. In contrast, this is easy to check with the inside-out order, in which the innermost sugar is expanded first, because this is akin to type checking function calls in programming languages and it's well-known how to accomplish this.

3. To be useful, the outside-in order requires the ability to arbitrarily rewrite a term. This is very powerful but dangerous at the same time. For instance, it is possible to create sugar that reverses all function argument lists inside of it. This would be misleading to language users because they might believe that the syntax they saw is an ordinary function application while it is not. Kiselyov demonstrated this problem in Scheme—whose macro system uses an outside-in expansion order—by breaking macro hygiene, a property that the language claims to have [4].

1.3 Our Contribution

We adapt the resugaring work by Pombrio et al. so that our resugaring system expands sugar in the inside-out order. Therefore, our thesis statement is:

> It is possible to create an inside-out resugaring system.

Additionally, we lift some restrictions in their work, making it applicable to a larger variety of syntactic sugar. In particular, a matched term could not be duplicated in their work. In practice, however, this occurs naturally. Our system hence supports term duplication. We find that our new resugaring system is sufficiently expressive while it avoids the disadvantages of desugaring in outside-in order, as described in Section 1.2.

We also implemented this system concretely for Pyret, an educational programming language. Pyret is used in several introductory computer science courses, including some of the Bootstrap curricula, which teach algebra, data science, and physics via programming [7]. As a language designed for introductory courses, Pyret is a perfect target to implement an algebraic stepper,

[6]Some programming languages that have outside-in syntax expansion introduces a construct to allow temporary inside-out expansion [1]. We could not see how we could integrate these kind of constructs while maintaining our resugaring system.

which will allow students to see how their programs evaluate step-by-step. We also converted most of Pyret's desugaring code which spans about 1000 line long to the new pattern-based desugaring which spans only about 500 line long, showing that the pattern-based desugaring rules can be expressed concisely. Our work was incremental, meaning that we can gradually migrate existing infrastructure to our new system. Moreover, our work didn't need to change Pyret's infrastructure much, showing that it can integrate well to an already existing language.

Chapter 2

A Desugaring System for Pyret

In this chapter, we describe the language we developed for writing desugaring rules for Pyret.

2.1 Terms

Pyret has approximately 150 different kinds of AST nodes, and this number will keep increasing as it has more features. A desugaring system that needs to be aware of all kinds of AST nodes would be both unmanageable and unsustainable. Our desugaring system thus works with *term*s, which are uniform structures that store AST node's information. Given an AST node, the node's type becomes a constructor name, and the node's children become a list of subterms. Since an AST component could be an option, a list, or a primitive value, they are reflected in our definition of a term as well.

Unfortunately, Pyret sometimes re-uses the same name for both syntactic sugar and core syntax. For example, `s-if-else` with multiple branches is considered syntactic sugar, whereas `s-if-else` with one branch is considered core syntax. For this reason, given a term, it is impossible to tell just from the constructor name whether it should be expanded. Our definition therefore explicitly distinguishes between *core term*s (`t-core`), which should not be expanded, and *surface term*s (`t-surf`), which should be expanded. The initial user-written program, which is in the surface language, then starts out as a *pure* surface term, meaning that all subterms of the initial term are surface terms (and not core terms).

For example, the following AST in Pyret:

```
s-if-else([list: s-if-branch(s-bool(true), s-num(1))], s-num(2))
```

can be expressed as a term as:

```
<term> ::= <primitive-term>                          ; primitive value (t-prim)
         | "(" <constructor-name> <term>* ")"        ; surface (t-surf)
         | "<" <constructor-name> <term>* ">"        ; core (t-core)
         | "{" <constructor-name> <term>* "}"        ; auxiliary (t-aux)
         | "[" <term>* "]"                           ; list (t-list)
         | "{" "some" <term> "}"                     ; option (t-option)
         | "none"                                    ; option (t-option)
         | <var-name>                                ; variable (t-var)

<primitive-term> ::= NUMBER
                   | STRING
                   | BOOLEAN
                   | SRCLOC
```

Figure 2.1: BNF Grammar for Pyret Terms

```
t-surf("s-if-else",
  [list:
    t-list([list:
      t-surf("s-if-branch",
        [list: t-surf("s-bool", t-prim(bool(true))),
      t-surf("s-num",  t-prim(num(1)))])]),
    t-surf("s-num", t-prim(num(2)))])
```

The full definition of terms is shown in Fig. 2.1. We only need 8 cases as opposed to 150 cases due to the representation of terms we described above.

With the above mechanism, desugaring is a process transforming a pure surface term to a pure core term. This will be described precisely in the next section.

During the desugaring process, we might desire to use a data structure beyond the syntax of the language itself. For example, it can be helpful for sugar to rewrite to or make use of a pair. These kind of structures can be represented as an *auxiliary* term. The auxiliary term obviously can't be in the initial term. Moreover, if desugaring succeeds, then the final term should be in the core language, meaning there should be no auxiliary terms left as well.

2.2 Desugaring

The surface language is large: it includes both the language's syntactic sugar and its core syntax. Thus, desugaring a surface term must deal with both of these cases. If the surface term matches a desugaring rule, then it is syntactic sugar, and will be rewritten to what it stands for, as defined by that rule. On the other hand, if a surface term does not match a desugaring rule, then it is core syntax. In this case, desugaring will implicitly convert it to a core term.

The grammar of the desugaring rule language is given in Fig. 2.2. Each *sugar* has a number of *cases*, each of which has a LHS pattern and a RHS pattern. Expanding a surface term whose sugar is defined proceeds by finding the first LHS that matches, and then rewriting it by using the RHS

```
<program> ::= <sugar>*
<sugar> ::= "sugar" <sugar-name> ":" ("|" <case>)* "end"
<case> ::= <pattern> "=>" <pattern>
<pattern> ::= <primitive-pattern>    ; primitive pattern
            | <pattern-var>          ; pattern variable
            | "_"                    ; wildcard
              ; surface pattern
            | "(" <constructor-name> <srcloc> <pattern>* ")"
              ; core pattern
            | "<" <constructor-name> <srcloc> <pattern>* ">"
              ; auxilliary pattern
            | "{" <constructor-name> <pattern>* "}"
            | "[" <pattern>* [<ellipsis>] "]"   ; list
            | "{" "some" <pattern> "}"          ; option
            | "none"                            ; option
              ; metafunction
            | "(" "meta" <meta-name> <pattern>* ")"
              ; metafunction (reversible)
            | "(" "biject" <biject-name> <pattern> ")"
              ; capture variable(s)
            | "(" "capture" "[" <var-item>* "]" <pattern> ")"
              ; fresh variable(s)
            | "(" "fresh" "[" <var-item>* "]" <pattern> ")"
            | <var-name>   ; variable

<primitive-pattern> ::= NUMBER
                      | STRING
                      | BOOLEAN
<srcloc> ::= ""            ; implicit srcloc (propogate top srcloc)
           | "@" <pattern> ; explicit srcloc
<pattern-var> ::= <pattern-var-name> ["_{" <label-name>* "}"] [":" <type>]
<ellipsis> ::= <pattern> "..." <label-name>     ; >= 0 repetitions
<var-item> ::= <var-name>                       ; intro of var
             | "[" <var-item> "..." <label-name> "]" ; intro of seq of vars
```

Figure 2.2: BNF Grammar for Pyret's Desugaring Language

```
if a:
  1
else if b:
  2
else:
  3
end
```

(a) A program before desugaring (as code)

```
if a:
  1
else:
  if b:
    2
  else:
    3
  end
end
```

(b) The program after desugaring (as code)

```
(s-if-else [(s-if-branch (s-id a) (s-num 1))
            (s-if-branch (s-id b) (s-num 2))]
           (s-num 3))
```

(c) The program before desugaring (as a term)

```
<s-if-else [<s-if-branch <s-id a> <s-num 1>)]
           <s-if-else [<s-if-branch <s-id b> <s-num 2>>]
                      <s-num 3>>>
```

(d) The program after desugaring (as a term)

Figure 2.3: Example of the if-else sugar

pattern as a template. Expanding a surface whose sugar is not defined proceeds by rewriting it to a core term as described above. This is how to expand *single surface term*. To desugar a whole term, our desugaring system will recursively expand all of its subterms using the inside-out order. Note that if sugar is defined but no LHS matches the surface term, then desugaring fails.

An example that we will demonstrate here is a desugaring of the aforementioned `s-if-else` construct. `s-if-else` allows multiple branches of `if` and `else if`, but desugars into nested if-else's, each with exactly one branch (see Fig. 2.3).

In our desugaring language, we can express this desugaring as follows:

```
sugar s-if-else:
  | (s-if-else [] else) => else
  | (s-if-else [branch rest_{i} ...i] else) =>
    <s-if-else [branch] (s-if-else [rest_{i} ...i] else)>
end
```

The first case says that if only the else branch exists (because the if branches are empty), bind the pattern variable `else` to the else branch term, and rewrite the whole term to simply `else`.

The second case says that there must be at least one if branch. Bind the pattern variable `branch` to the first branch, and bind the pattern variable `rest` to the rest of the branches (using ellipses which informally mean "match zero or more terms"). The rewrite proceeds by recursively invoking the `s-if-else` sugar (with branches as `rest`). Then, create a core `s-if-else` that has `branch` as the only branch, and the result of the expansion of recursive sugar as the else branch.

2.3 Patterns

In this section, we describe our pattern language.

A pattern can appear as the LHS or RHS of a rule. When it appears as the LHS, it is used for matching against a surface term. When it appears as a RHS, it is used as a template to rewrite to. The semantics for the RHS is often simply a counterpart of LHS, so we might be not explicitly explain it if it's straightforward.

2.3.1 Primitives

On the LHS, a primitive pattern like `3` literally matches only a primitive term `3`. On the RHS, a primitive pattern like `3` produces a primitive term `3`.

2.3.2 Pattern variables

On the LHS, a pattern variable without label p matches any term and binds p to the term it matches. If there are multiple occurrences of p, every term p matches must be the same term. Otherwise, the matching fails. On the RHS, it rewrites to the term it matched on the LHS.

For example, consider:

```
sugar A:
  | (A x x) => <B x>
end
```

With the above sugar `A`, `(A <C 42> <C 42>)` expands to `<B <C 42> <C 42>`, whereas `(A <C 1> <C 2>)` fails to expand.

For labeled pattern variable, see Section 2.3.7

2.3.3 Surface and Core

On the LHS, a surface pattern $(C\ p_1\ \ldots\ p_n)$ or a core pattern $\langle C\ p_1\ \ldots\ p_n\rangle$ matches a surface term $(C\ t_1\ \ldots\ t_n)$ or a core term $\langle C\ t_1\ \ldots\ t_n\rangle$ respectively as long as p_i matches t_i for $1 \leq i \leq n$. Due to our inside-out expansion order, a surface pattern can only appear as the outermost construct in the LHS. It doesn't make sense for a surface pattern to appear nested inside the LHS pattern because all children of the outermost surface term would have already been expanded to core terms, so matching against them will definitely fail. A core pattern can appear on the LHS to match a core term, although this is strongly discouraged because it allows the system to arbitrarily deconstruct and rewrites terms, as mentioned in Section 1.3. It is encouraged to rewrite terms to an auxiliary structure and match against the auxiliary structure instead.

On the RHS, a surface or core pattern creates a surface or core term respectively. Unless explicitly stated otherwise, a surface and core term created this way will inherit the source location of the expanding surface term.

To explicitly bind a source location on the LHS, use `@l` where `l` is a pattern variable. To explicitly put a source location into a core or surface term on the RHS, use `@p` where `p` is any pattern that rewrites to a source location. For example:

```
sugar A:
  | (A @l 0) => <B l>
  | (A 1) => <B 1>
  | (A @l 2) => <B @(C l) 2>
    # assuming that (C l) rewrites to a source location
end
```

Here:

- `(A &location1 0)` expands to `<B &location1 &location1>`.
- `(A &location2 1)` expands to `<B &location2 1>`.
- `(A &location3 2)` expands to `<B &location 2>` where `&location` is whatever `(C &location3)` rewrites to.

2.3.4 Auxiliaries

On the LHS, an auxiliary pattern $\{C\ p_1\ \ldots\ p_n\}$ matches an auxiliary term $\{C\ t_1\ \ldots\ t_n\}$ as long as p_i matches t_i for $1 \leq i \leq n$.

2.3.5 Wildcards

A wildcard pattern, written as `_`, can only appear on the LHS. It matches any term and simply ignores it.

2.3.6 Lists (without ellipses)

On the LHS, a list pattern $[p_1\ p_2\ \ldots\ p_n]$ matches a list term $[t_1\ t_2\ \ldots\ t_n]$ as long as p_i matches t_i.

2.3.7 Lists (with ellipses)

On the LHS, a list pattern $[p_1\ p_2\ \ldots\ p_n\ p \cdots_l]$ matches a list term $[t_1\ t_2\ \ldots\ t_{n+k}]$ where $k \geq 0$ as long as p_i matches t_i for $1 \leq i \leq n$, and p matches t_i for $n < i \leq n + k$. By default, a pattern variable within p must bind to the *same* term in each t_i for $n < i \leq n + k$. However, if the pattern variable is labeled with l (the label on the ellipsis $\cdots_l$), then it can bind to distinct terms in each t_i. For example:

```
sugar A:
  | (A x [[x z_{l} 1] ...l]) => <B x [z_{l} ...l]>
end
```

With the above example, `(A 7 [[7 "a" 1] [7 "b" 1] [7 "c" 1]])` expands to `<B 7 ["a" "b" "c"]>`. Note, however, that if we change `z_l` to `z` in the above rule, the matching would fail because `"a"`, `"b"`, and `"c"` are distinct.

2.3.8 Fresh, Capture, and Variables

Fresh, capture, and variable patterns can only appear on the RHS. The sole purpose of fresh and capture patterns is to create a variable[1].

Often, we would like to create a temporary variable. The temporary variable's name should not clash with other variables' names. Fresh patterns implement exactly this mechanism: a fresh variable will expand to an actual name that is different from any other existing variable names. For example, the `or` sugar in the first chapter would be written as follows in the desugaring language:

```
sugar s-or:
  | (s-or a b) =>
    (fresh [x]
      (s-let x a (s-if-else [(s-if-branch (s-id x) x)]
                            b)))
end
```

In the above example, even though the pattern variable `a` or `b` might match against a Pyret variable `x`, because of the fresh pattern, the variable `x` that is used inside the fresh pattern will be a different variable.

In some situations, however, we would like to intentionally introduce a captured variable. This is useful for implementing magically bound identifiers like `this` and `self`. Capture patterns implement this mechanism: a captured variable will have its name identical to what is given. Pyret in fact does not currently need this feature, but we added it since it may be useful in the future.

Fresh and capture patterns also support creating multiple variables at once, and creating a series of variables using an ellipsis. Note that fresh and capture patterns cannot shadow a variable which was already introduced by another fresh or capture patterns.

2.3.9 Metafunctions and Bijections

Sometimes, we might want to use Pyret to rewrite a term because:

- Our desugaring language does not have an ability to access a property. For example, it is not possible to access a variable name as a string.
- Rewriting in Pyret might be easier. For example, it is possible to reverse a list term in our language, but it is not ideal at all. Doing so involves creating a helper sugar to fold over the list, which is tedious and less efficient.

For these reasons, we introduce *metafunction*s and *bijection*s to our language. A metafunction is simply a function written in Pyret that consumes terms and produces a term. There is no restriction for what the function could be, but the resugaring framework will not be able to resugar a core term produced by the metafunction back to a surface term. Therefore, to allow resugaring, we must make sure that all terms in the LHS can be reconstructed from the information from the RHS without using

[1] In Pyret, a variable name corresponds to `Name` in the AST definition.

the output of the metafunction. On the other hand, a bijection is two Pyret functions from term to term. These two functions must be inverses of each other for resugaring to work correctly. Because the functions are invertible, resugaring can resugar a core term produced by a bijection back to a surface term.

Writing a metafunction (or a bijection) must be done carefully. In particular, terms that a metafunction consumes might be tagged with additional information from the original program as described in Section 1.1. Hence, metafunctions should preserve the tags so that resugaring can be performed.

After implementing a metafunction or bijection in Pyret, it can be invoked by using a metafunction or a bijection pattern, which can only appear on the RHS. For example:

```
sugar s-rev-app:
  | (s-rev-app f args) => (s-app f (biject reverse args))
end
```

Chapter 3

Internals

In this chapter, we describe the internals of our desugaring and resugaring system.

3.1 Overview

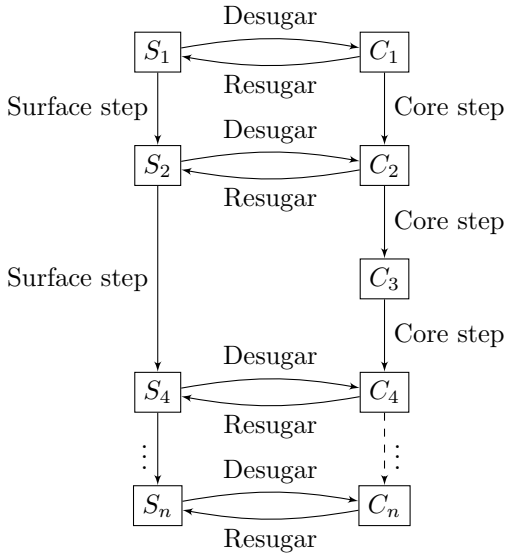

Figure 3.1: Evaluation step resugaring

Desugaring, as described in the previous chapters, is a transformation from a term in the surface language, fully written by language users, to a term in the core language where some subterms might be tagged with some additional information for resugaring. We call a term that is tagged with additional information a *tag term*, which will be described later in this chapter. Formally, we let $desugar(t) = t'$ denote that t desugars to t'.

In the context of evaluation step resugaring, *resugaring* is the inverse of desugaring. It is a

transformation from a term in the core language to a term in the surface language. In contrast to desugaring, resugaring could *fail* if there is no suitable surface representation for the core term. For example, C_3 in Fig. 3.1 and the third step in Fig. 1.2b have no corresponding surface step. The causes of the failure will be explained in Section 3.2.2. Formally, we let $resugar(t') = t$ denote that t' resugars to t.

For resugaring to be a proper inverse of desugaring, we need the following two properties:

- For all user-written terms t in the surface language, if $desugar(t) = t'$, then $resugar(t') = t$.
- For all terms t' in the core language, if $resugar(t') = t$, then $desugar(t) = t'$.

Additionally, resugaring should present evaluation steps in terms of the syntax that the user wrote.

3.2 Matching, Substitution, and Specialization

Similar to how desugaring is a series of expansions in inside-out order, we also choose resugaring to be a series of *unexpansions* in inside-out order. However, both expansions and unexpansions can be described in terms of more elementary operations: *matching*, *substitution*, and *specialization*.

Given a term t and a pattern p, we can try matching t against p, denoted by t/p. If the matching succeeds, it will produce an *environment* γ, which contains bindings for pattern variables. Because t could be a tag term, which is an internal structure that clients should need to know about, matching should be performed on the term component inside the tag term instead.

Similar to matching, given an environment γ and a pattern p', we can try substituting γ into p', denoted by $\gamma \bullet p'$. If the substitution succeeds, it will produce a term t'.

With the above operations, the expansion of a term t with a desugaring rule's case $p \to p'$ is $exp(t, p \to p') = (t/p) \bullet p'$, assuming that matching and substitution succeed.

It might be tempting to think that an unexpansion of a term t' is $(t'/p') \bullet p$. That is to run matching and substitution in reverse. However, there are a few issues:

- Some pattern variables from the LHS might not occur in the RHS. Therefore, in substitution during unexpansion, these pattern variables would be unbound.
- It is possible that there will be several desugaring rule's cases that have similar RHSs, so there is no unique p' and p in the unexpansion direction.
- If t is a tag term that matched against p during expansion, then the unexpansion $(t'/p') \bullet p$ would not restore information in the tag term.

We circumvent this problem by tweaking the above process by adding an additional phase called *specialization*. When an expansion of t with a case $p \to p'$ succeeds, resulting in a term t', a specialization of t with $p \to p'$ results in (p_{spz}, p'_{spz}) such that $(t'/p'_{spz}) \bullet p_{spz} = t$. In particular,

the problem about dropped pattern variables is solved by computing what information are dropped, and replacing the dropped pattern variables in p with this information, resulting in p_{spz}. The problem about lost tags can be solved in a similar fashion. Note that we have implemented, but not formalized, this specialization phase.

Now that we have p_{spz} and p'_{spz}, we need to store them to make resugaring possible. We employ tag terms for this purpose.

3.2.1 Tag Terms

To progagate p_{spz} and p'_{spz} obtained during an expansion from t to t', we tag both p_{spz} and p'_{spz} to t'. Formally, we write a tag term for t' with p_{spz} and p'_{spz} as $Tag_{p_{spz} \Rightarrow p'_{spz}}(t')$. Thus, to propagate p_{spz} and p'_{spz} is to use $Tag_{p_{spz} \Rightarrow p'_{spz}}(t')$ instead of simply t'.

3.2.2 Tracking User's Code

In Fig. 1.2a, we see that the step `(not (if #f #f (not #f)))` in the core language has no corresponding step in the surface language. However, it turns out that a series of unexpansions will successfully unexpand all tags in this term, and the result exposes the core syntax of `or`, which is undesirable, because resugaring is supposed to produce a term whose components were written by language users.

We circumvent this by tracking whether components of the "resugared" term (subterms) were written by the user or not, and rejecting any "resugared" term that contains non-user-written one.

3.2.3 Environment

An environment has two components. A *pattern variable map* is a dictionary that maps each pattern variable to the term it matched. An *ellipsis map* is a dictionary that maps each ellipsis label to a list of environments. Formally, we write $\gamma = \{\alpha_1 \mapsto t_1, \ldots, \alpha_n \mapsto t_n, l_1 \mapsto \lceil \gamma_1 \ldots \rceil, \ldots, l_m \mapsto \lceil \gamma_m \ldots \rceil\}$ to represent a pattern variable map of size n and an ellipsis map of size m. Note that we use a different notation for these lists simply to distinguish them from list terms/patterns.

Often, we will need to unify multiple environments together. As long as there is no conflict (e.g., trying to unify $\{\alpha_1 \mapsto 1\}$ and $\{\alpha_1 \mapsto 2\}$), unification will succeed. However, if there is a conflict, the unification will fail.

3.2.4 Matching

The formal semantics for matching is given at Fig. 3.2[1].. During matching, we need to additionally accumulate fresh variables into a *fresh set*, which starts as an empty set. The role of the fresh set is to ensure that all occurrences of a fresh variable are indeed the same. This only matters in the matching during unexpansion, since fresh patterns can only appear in the RHS.

[1]Both formal semantics for matching and substitution are joint work with Justin Pombrio.

$\boxed{F \vdash t/p = \gamma}$

$$\text{m-pvar}\ \frac{}{F \vdash t/\alpha = \{\alpha \mapsto t\}} \qquad \text{m-var-capture}\ \frac{x \notin F}{F \vdash x/x = \{\}}$$

$$\text{m-wildcard}\ \frac{}{F \vdash t/\_ = \{\}} \qquad \text{m-var-fresh}\ \frac{x \in F}{F \vdash y/x = \{coerce(x) \mapsto y\}}$$

$$\text{m-prim}\ \frac{}{F \vdash prim/prim = \{\}} \qquad \text{m-capture}\ \frac{F \vdash t/p = \gamma}{F \vdash t/(Capture\, x.\, p) = \gamma}$$

$$\text{m-empty}\ \frac{}{F \vdash [\,]/[\,] = \{\}} \qquad \text{m-fresh}\ \frac{F, x \vdash t/p = \gamma}{F \vdash t/(Fresh\, x.\, p) = \gamma}$$

$$\text{m-cons}\ \frac{\begin{array}{c} F \vdash t_1/p = \gamma_1 \\ F \vdash [t_2 \ldots t_n]/[ps] = \gamma_s \\ \gamma_1 \cup \gamma_s = \gamma \end{array}}{F \vdash [t_1 \ldots t_n]/[p\ ps] = \gamma} \qquad \text{m-biject}\ \frac{F \vdash f^{-1}(t)/p = \gamma}{F \vdash t/(biject\ f\ p) = \gamma}$$

$$\text{m-meta}\ \frac{}{F \vdash t/(meta\ f\ p_1 \ldots p_n) = \{\}}$$

$$\text{m-ellipsis}\ \frac{F \vdash t_1/p = \gamma_1\ \ldots\ F \vdash t_n/p = \gamma_n}{F \vdash [t_1 \ldots t_n]/[p \cdots \imath] = \{l \mapsto check(\lceil \gamma_1 \ldots \gamma_n \rceil, l)\}}$$

$$\text{m-core}\ \frac{F \vdash t_1/p_1 = \gamma_1\ \ldots\ F \vdash t_n/p_n = \gamma_n}{F \vdash \langle C\ \ t_1 \ldots t_n \rangle / \langle C\ \ p_1 \ldots p_n \rangle = \gamma_1 \cup \ldots \cup \gamma_n}$$

$$\text{m-aux}\ \frac{F \vdash t_1/p_1 = \gamma_1\ \ldots\ F \vdash t_n/p_n = \gamma_n}{F \vdash \{C\ \ t_1 \ldots t_n\} / \{C\ \ p_1 \ldots p_n\} = \gamma_1 \cup \ldots \cup \gamma_n}$$

$$\text{m-tag}\ \frac{F \vdash t/p = \gamma}{F \vdash Tag_{p_{lhs} \Rightarrow p_{rhs}}(t)/p = \gamma}$$

Figure 3.2: Matching

Similarly, bijection pattern can only appear in the RHS, so it only matters in the matching during unexpansion. For this reason, the pattern computes $f^{-1}(t)$ rather than $f(t)$.

The function $check(\lceil \gamma_1 \dots \gamma_n \rceil, l)$ which is used in the `m-ellipsis` case checks that pattern variables that are not labeled with l have the same value. If the check fails, the matching will fail as well. Otherwise, the function simply returns $\lceil \gamma_1 \dots \gamma_n \rceil$.

The function $coerce(x)$ which is used in the `m-var-fresh` case converts a variable into a pattern variable that will not clash with existing pattern variables. We use it in `m-var-fresh` as a way to make sure that during unexpansion, variable names originated from a fresh pattern will be the same.

3.2.5 Substitution

The formal semantics for substitution is given at Fig. 3.3. Similar to matching, during substitution, we need to additionally accumulate variables into the variable map, which starts as an empty map. The role of the variable map is to keep track of variable names so that the fresh pattern can create a new fresh name correctly.

Now that we have matching, substitution, and specialization, we also obtain desugaring and resugaring. The only component left is the algebraic stepper.

3.3 Algebraic Stepper

After a program is desugared, we have a program in the core language. To implement an algebraic stepper for the core language is to instrument code such that at runtime, the program shows evaluation steps in addition to evaluating to a value.

Implementing this can be done by *quoting* the program at compile time, which turns terms into syntax objects in the runtime, and passing the quoted program to a *stepify* function. The stepify function then will interpret the code as well as emitting evaluation steps.

To resugar evaluation steps, all needed to be done is to invoke the `resugar` function for every emitting steps and only show steps that successfully resugar.

$\boxed{V \vdash \gamma \bullet p = t}$

$$\text{s-pvar}\ \frac{\alpha \mapsto t \in \gamma}{V \vdash \gamma \bullet \alpha = t} \qquad \text{s-var}\ \frac{x \mapsto y \in V}{V \vdash \gamma \bullet x = y}$$

$$\text{s-prim}\ \frac{}{V \vdash \gamma \bullet prim = prim} \qquad \text{s-fresh}\ \frac{V, (x \mapsto y) \vdash \gamma \bullet p = t \quad \text{for fresh } y}{V \vdash \gamma \bullet (Fresh\, x.\, p) = t}$$

$$\text{s-empty}\ \frac{}{V \vdash \gamma \bullet [\,] = [\,]} \qquad \text{s-capture}\ \frac{V, (x \mapsto x) \vdash \gamma \bullet p = t}{V \vdash \gamma \bullet (Capture\, x.\, p) = t}$$

$$\text{s-cons}\ \frac{\begin{array}{c} V \vdash \gamma \bullet p = t_1 \\ V \vdash \gamma \bullet [ps] = [t_2 \ldots t_n] \end{array}}{V \vdash \gamma \bullet [p\ ps] = [t_1 t_2 \ldots t_n]} \qquad \text{s-biject}\ \frac{V \vdash \gamma \bullet p = t}{V \vdash \gamma \bullet (biject\ f\ p) = f(t)}$$

$$\text{s-meta}\ \frac{V \vdash \gamma \bullet p_1 = t_1\ \ldots\ V \vdash \gamma \bullet p_n = t_n}{V \vdash \gamma \bullet (meta\ f\ p_1 \ldots p_n) = f(t_1, \ldots, t_n)}$$

$$\text{s-ellipsis}\ \frac{\begin{array}{c} l \mapsto \lceil \gamma_1 \ldots \gamma_n \rceil \in \gamma \\ V \vdash \gamma \cup \gamma_1 \bullet p = t_1\ \ldots\ V \vdash \gamma \cup \gamma_n \bullet p = t_n \end{array}}{V \vdash \gamma \bullet [p \cdots_l] = [t_1 \ldots t_n]}$$

$$\text{s-ellipsis-list}\ \frac{\begin{array}{c} l \mapsto \lceil \gamma_1 \ldots \gamma_n \rceil \in \gamma \\ V \vdash \gamma \cup \gamma_1 \bullet p_1 = t_1\ \ldots\ V \vdash \gamma \cup \gamma_n \bullet p_n = t_n \end{array}}{V \vdash \gamma \bullet \lceil p_1 \ldots p_n \rceil = [t_1 \ldots t_n]}$$

$$\text{s-core}\ \frac{V \vdash \gamma \bullet p_1 = t_1\ \ldots\ V \vdash \gamma \bullet p_n = t_n}{V \vdash \gamma \bullet \langle C\ \ p_1 \ldots p_n \rangle = \langle C\ \ t_1 \ldots t_n \rangle}$$

$$\text{s-surf}\ \frac{V \vdash \gamma \bullet p_1 = t_1\ \ldots\ V \vdash \gamma \bullet p_n = t_n}{V \vdash \gamma \bullet (C\ \ p_1 \ldots p_n) = (C\ \ t_1 \ldots t_n)}$$

$$\text{s-aux}\ \frac{V \vdash \gamma \bullet p_1 = t_1\ \ldots\ V \vdash \gamma \bullet p_n = t_n}{V \vdash \gamma \bullet \{C\ \ p_1 \ldots p_n\} = \{C\ \ t_1 \ldots t_n\}}$$

$$\text{s-tag}\ \frac{V \vdash \gamma \bullet p = t}{V \vdash \gamma \bullet Tag_{p_{lhs} \Rightarrow p_{rhs}}(p) = Tag_{p_{lhs} \Rightarrow p_{rhs}}(t)}$$

Figure 3.3: Substitution

Chapter 4

Conclusion

4.1 Evaluation

We created the inside-out resugaring system in Pyret and run test cases to test our system against an algebraic stepper for a call-by-value lambda calculus with numbers and booleans (and their basic operations like conditional constructs). These test cases cover all kinds of patterns in our pattern-based languages. All of these tests except `cond` in Racket work as expected. `cond` technically works as expected too, since it abides the correctness properties in Section 3.1. However, most algebraic steppers show some steps that are deemed incorrect by our system as these steps are intuitive to show. Pombrio et al.'s system has a feature in their pattern-based language to break the correctness properties. It is straightforward to support such feature too, although we have not implemented it due to our time constraints.

One issue that came up in both Pombrio et al.'s and our work is that if a language developer chooses to implement identifier lookup by using an environment, the steps could appear to contain unbound identifiers, whereas if the language developer uses substitution instead, this would not be a problem. We considered this a user interface problem, and due to time constraints, we did not attempt to fix the problem. A possible solution includes displaying the environment in the algebraic stepper as well, but this might not be preferable. Since most interpreters use an environment to implement identifier lookup, it remains an open, interesting question for how to present the result of the algebraic stepper in an intuitive way.

The result of the evaluation supports our thesis statement, which is that it is possible to create an inside-out resugaring system.

4.2 Ongoing Work

Although we finished creating an inside-out resugaring system, we have not finished creating the algebraic stepper and its graphical user interface for the whole Pyret language yet due to time

constraints. We also have not implemented the grammar checker, although our system was designed to make this straightforward to implement.

4.3 Future Work

Our resugaring system is carefully designed to allow some features that Pyret tentatively plans to support in the future. These include:

- Pyret's well-formedness checker, which prohibits Pyret programs that are considered ill-formed, could be built on top of our pattern-based desugaring language (see more information in Appendix A). It remains an open question how to modify the system to collect well-formedness errors as many as possible before showing them to users.

- By exposing our pattern-based desugaring language to Pyret users, we could allow users to add a new syntax and restrict how existing syntax could be used. Due to the inside-out expansion order, it is guaranteed that the added syntax will not interfere with already existing syntax in a way that makes language users be mislead.

  The ability to add non-interfering syntax and restrict already existing syntax lets users to create a language level, which is a customized language derived from the base language (Pyret, in this case). One of its use is for teaching students in a custom setting so that the language is more intuitive to them.

Bibliography

[1] 12.4 syntax transformers. https://docs.racket-lang.org/reference/stxtrans.html#%28def._%28%28quote._~23~25kernel%29._local-expand%29%29.

[2] ISO/IEC 9899:TC2.

[3] John Clements, Matthew Flatt, and Matthias Felleisen. Modeling an algebraic stepper. In *Proceedings of the 10th European Symposium on Programming Languages and Systems*, ESOP '01, pages 320–334, London, UK, UK, 2001. Springer-Verlag.

[4] Oleg Kiselyov. How to write seemingly unhygienic and referentially opaque macros with syntax-rules. 10 2002.

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

[6] Justin Pombrio and Shriram Krishnamurthi. Resugaring: Lifting evaluation sequences through syntactic sugar. pages 361–371, New York, NY, USA, 2014. ACM.

[7] Emmanuel Schanzer, Kathi Fisler, Shriram Krishnamurthi, and Matthias Felleisen. Transferring skills at solving word problems from computing to algebra through bootstrap. In *Proceedings of the 46th ACM Technical Symposium on Computer Science Education*, SIGCSE '15, pages 616–621, New York, NY, USA, 2015. ACM.

Appendix A

Sugar in Pyret

In this appendix, we describe components related to sugar in Pyret's compilation process after we added our system. It can be read as a reference manual for how to add a new sugar to Pyret.

AST definition

Pyret AST definition is located at `src/arr/trove/ast.arr`. Adding a sugar requires adding an AST node case to an appropriate data definition along with method `label` and `tosource` (for pretty-printing). For example, suppose the `table` sugar doesn't exist yet and we want to add it. Because it's a Pyret expression, it should be added under the data definition `Expr`.

Additionally, AST visitors and the AST-term converter (see step C. and G. in Fig. A.1) need to reflect the change to the AST. We made this easier by using metaprogramming to read the AST definition file and auto-generate both AST visitors and the AST-term converter. To auto-generate them, run `node build/phase0/pyret.jarr -auto-generate`.

One of the current limitations of the metaprogramming is that every primitive value in an AST node can be only either a number, a string, a boolean, or a source location. A compound value can be only either an AST node, a list of values, or an option of a value. These suffice for the current state of Pyret. However, if there is a need to support a wider variety of values in the future, the metaprogramming needs to change. The relevant files include: `src/arr/compiler/autogenerate.arr` which defines how generated files should be written, and `src/arr/compiler/visitors/ast-transformer.arr` which defines the AST definition reader.

Parser

Adding a sugar requires changing both the Pyret grammar and the translation from parse tree to AST. The Pyret grammar, located at `src/js/base/pyret-grammar.bnf`, is written in Backus–Naur form (BNF). The compilation process will automatically generate a parser based on this file. The translator, on the other hand, is located at `src/js/trove/parse-pyret.js`.

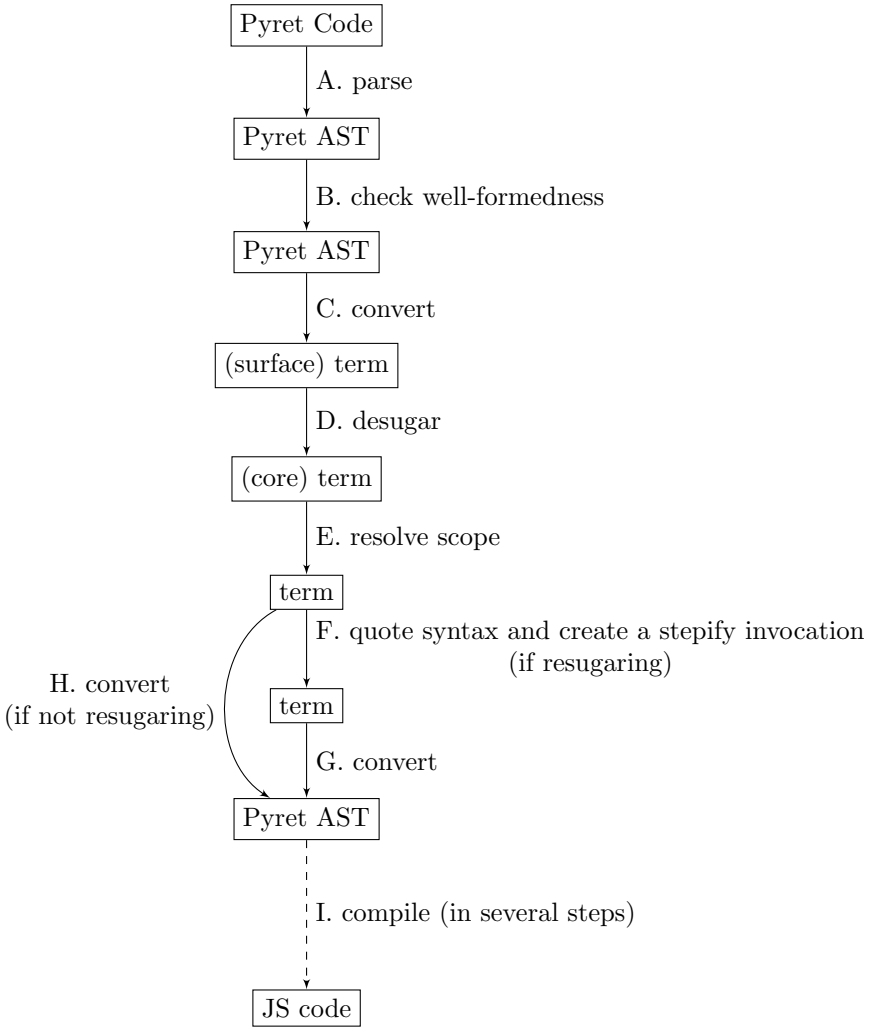

Figure A.1: The Pyret compilation pipeline with our system

Well-formedness checking

Since BNF is context-free, Pyret's grammar won't be able to prohibit ill-formed programs for context-sensitive well-formedness criteria. An example is the object syntax which should not have duplicate field names. For this reason, Pyret has well-formedness checking to disallow these ill-formed programs. Moreover, while some ill-formed programs could be prohibited by the grammar, the resulting parse error would be rather uninformative. We hence might want to allow an ill-formed program to parse and catch the ill-formed program during the well-formedness checking so that we can report a more informative error.

The well-formedness checker is defined at `src/arr/compiler/well-formed.arr`. Adding sugar might or might not require changing the well-formedness checker depending on whether the sugar needs to check its well-formedness.

In the future, we have a plan to integrate the well-formedness checker as a part of the desugaring language (see Section 4.3).

AST-term converter

In order to use the pattern-based desugaring language, we need a way to manipulate terms without needing to be aware of the Pyret AST structure. For this reason, we define a data definition `Term` to represent a term (see Section 2.1). As mentioned before, the converter between Pyret AST and `Term` could be done by running `node build/phase0/pyret.jarr -auto-generate`. The conversion makes sure that all terms have a source location by adding a dummy location to them if they don't already have one.

Desugaring

Desugaring rules can be written in the pattern-based desugaring language described in Chapter 2. The definition of desugaring rules is tentatively located at `src/arr/trove/pyret.sugar`.

Algebraic Stepper

The algebraic stepper, as described in Section 3.3, is tentatively located at `src/arr/trove/stepper.arr`.