The Complex Essence of “Scope as a Preorder”

Summer Internship Report

by Lu Xiaoyang

Foreword

This article is to record my research experience in 2024 summer—from June 17 to August 10—at Programming Languages Lab, Peking University. The result of the research is quite trivial, lacking insights or key observations; however, the research process is rather complete. This article does not mean to be a technical report or a research paper, but I will try to demonstrate my thoughts and effort.

In this article, “I” and “we” refer to the author, “they” refers to [PKW17] or its authors, and “you” refers to the reader.

1 Osazone

Defining new languages on top of a general-purpose language through *syntactic macros* is a proven way to crafting domain specific languages (DSLs) quickly; one of the most prominent examples is Racket (along with its predecessors (really?) Common Lisp and Schene). The macros can be hygiene or not, Turing-complete or not, first-class or not, but they all transform the syntax (tree) to another one. By treating the DSL as simply a rephrasement of a powerful core language, you get the compiler, the runtime, the type checker, and so on for free.

But there’s no free lunch. Since we still do all the things on the core language, the use experience of a DSL defined in such a way is tightly coupled with the core language. For example, the error messages might be phrased in core-language constructs; the debugger (if any) has to be used on the core language; the semantics of the DSL has to be understood as a combination of macros and the semantics of the core language. No wonder Racket provides a macro stepper, not for language engineers only, but also for users.

There has been some work on relieving this situation. [PK14], [PK15] and [YXGH22] derive a evaluation sequence (like that in small-step operational semantics) from the core-language sequence together with information from macro expanding. This “resugaring” is of great help to understand the semantics of the DSL. Osazone, the project the research group here had been working on, chose a different route. While Osazone still occupies simple macros (in some terminology, *sugars*) in the style of [Fel91][1] to define new languages (of course this would rule out some complex macro systems that interleave macro-expanding and evaluation), it *lifts* the interpreter rather than reduce the program. Similar ideas appeared in [PK18] and [PKW17]. Ideally, understanding the derived interpreter should not require knowledge beyond the DSL.

That is the big picture, but the reality is still far from the ultimate goal.

1. As I mentioned previously, the macros supported is very limited. They must be in the form

   $$P(\bullet_1, \bullet_2, \ldots, \bullet_n) \ \rightarrow\ F(\bullet_1, \ldots, \bullet_n)$$

   Here the right hand side is a syntax tree with *holes* $\bullet_1,\ldots,\bullet_n$. The holes here must be used linearly, without duplication. The definition means that the subterm of P shall be extracted and put into holes in the RHS accordingly. We cannot inspect into the subterms; they are opaque. Considering that R^{6}RS introduces `syntax-case` to overcome the (already stronger than Osazone) expressivity of `syntax-rules`, we must have lost something interesting.

2. The theoretical foundation is not quite solid. The algorithm works basically by doing sort of partial evaluation, or specialization, or normalization on the RHS `eval`$(F(\bullet_1, \ldots, \bullet_n))$. Currently we can only explain when it works, and give some constraints; we do not come up with some novel method to overcome the limitation of partial evaluation. The good thing is that Osazone accepts anything that compute a denotation, including syntax-directed typing rules.

3. Currently, only interpreters can be lifted. If the user does not like to read the source code of the interpreter, Osazone offers no more than the expand-and-evaluate approach. Lifting error messages and debuggers mentioned previously, or friendly interface such as parsers and pretty-printers, are future research topics.

1.1 Architecture

Osazone is written in 4000 lines of pure Haskell. I shall briefly annotate what each part of the source code does. You can find this on `https://github.com/vbcpascal/Osazone-oopsla24`. Much code is devoted to infrastructures such as lexer, parser, pretty-printer, static checker, and so on; the core algorithm is in the file `Lifting/Lifting/SugarLifting.hs`.

```
├── Config ; parser for configuration files
│   ├── Lifting ; configuration of syntactic sugars
│   ├── Osazone ; configuration of language and program definition
│   ├── Review.hs ; for artifact review
│   └── YamlReader.hs
├── Language
│   ├── Dependency ; redundant
│   ├── Osazone ; the meta-language; very Haskell-like, and ‘compile’ to Haskell
│   │   ├── AST.hs ; abstract syntax tree
```

1. In practice, pattern-based macros in the style of [PKW17]—or `syntax-rules` without lists—may also work.

```
│   │   ├── Parser ; parser & lexer
│   │   ├── Pass ; static checking/transformation
│   │   │   ├── Application.hs ; wrapper
│   │   │   ├── ImportRenaming.hs ; add implicit qualifier to imported identifiers
│   │   │   ├── NameResolution.hs ; decide an identifier refers to which definition
│   │   │   ├── Rename.hs ; rename variables with fresh names; useful in macro expansion
│   │   │   ├── Simplify ; equivalent transformations to generate more compact code
│   │   │   ├── Simplify.hs ; wrapper
│   │   │   └── Standardization.hs ; process implicit monads
│   │   └── Pass.hs ; wrapper
│   └── Osazone.hs ; wrapper
├── Lifting
│   ├── Extension.hs ; structure of extension file, which contains filter, redefinition, and sugars
│   ├── Filter.hs ; filter out core-only constructs
│   ├── Lifting ; core algorithm
│   ├── Lifting.hs ; wrapper
│   ├── Parser.hs ; parser of sugar definition
│   └── Sugar.hs ; AST of sugar definition
├── Target
│   ├── Haskell ; dump runnable Haskell code
│   └── Operational ; dump deduction rules in HTML
└── Utils ; utility
```

2 Inferring Scope

In this section, I shall summarize the main ideas in [PKW17]; I suggest you to read the original text to fully understand their work. Then, I shall show various errors and their effects in that paper.

2.1 What they did

Almost every programming language has a concept of "variables". Roughly speaking, scoping, or binding, defines which variable refers to which. For example, in which part of the program can we access a declared variable? When we have two declaration with a same name, does one of them overrides (or "shadows") the other, or it is just illegal? This paper does three things:

1. Formalize precisely what scope means in a new way.
2. Give a binding language—that is, a compact way to specify how scope show work—base on the formalization.
3. Give a algorithm to automatically infer binding rules in the language for new constructs introduced by syntactic macros.

As we shall see, we will challenge the first, then extend the second and the third.

We work on (unityped) abstract syntax trees (ASTs). The syntax is shown in the following. The first line is variable definition, and the second line is variable reference. The last line on appears in macro definitions.

$$\begin{array}{rcl} t & \to & x^{\mathsf{D}} \\ & | & x^{\mathsf{R}} \\ & | & P(t_1, \ldots, t_n) \\ & | & \bullet_i \end{array}$$

They formalize scopes in a term as preorders over x^{D}s and x^{R}s, where the latter are least. Than we define $x^{\mathsf{R}} \mapsto x^{\mathsf{D}}$, or x^{R} is bound by x^{D}, to be

$$x^{\mathsf{D}} \in \{y \,|\, x^{\mathsf{R}} \leqslant y \wedge (\nexists z, x^{\mathsf{R}} \leqslant z \wedge y \nleqslant z)\}.$$

Notice that the RHS can contain more than one element. This formalization is in some sense equivalent to "binding as sets of scopes" ([Fla16]). We do not elaborate on it here, just comment that for each preorder $(S, \leqslant)$ there is a homomorphism to $(\mathcal{P}(S), \subseteq)$ by taking downward sets $x \mapsto \{y \,|\, y \leqslant x\}$.

How do we get such a preorder? They choose to specify for each constructor P a local preorder, and compose them in a term by taking a transitive reflexive closure. It turns out that this is not expressive enough. So they decide to embed into a richer preorder: associate each term with two *ports* $\uparrow t$ and $\downarrow t$, and defint $a \leqslant b \triangleq \downarrow a \leqslant \downarrow b$. Only the following subset is permitted for each P:

$$\begin{array}{ll} \downarrow t_i \leqslant \uparrow t_j & (\textsf{bind } j \textsf{ in } i \in \Sigma(P)), \\ \downarrow t_i \leqslant \downarrow P & (\textsf{import } i \in \Sigma(P)), \\ \uparrow P \leqslant \uparrow t_i & (\textsf{export } i \in \Sigma(P)), \\ \uparrow P \leqslant \downarrow P & (\textsf{re-export} \in \Sigma(P)), \end{array}$$

plus a fixed rule $\uparrow x^{\mathsf{D}} \leqslant \downarrow x^{\mathsf{D}}$ to ensure transitivity. Intuitively, as their names suggest, you can think of them as exported names and imported names. The following is an example directly pasted.

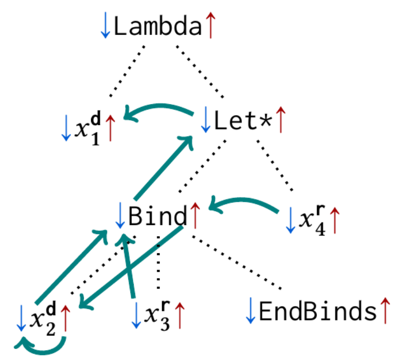


Figure 1. Illustration of a complete term decorated with a preorder

How to infer scope rules? First, note that we can define correctness of inference clearly: $x^{\mathsf{R}} \mapsto x^{\mathsf{D}}$ holds before macro expansion iff it holds after. This is in contrast to other studies where hygiene is difficult to formalize, if it is in any way formalized.[2] Either side of a macro (c_1, c_2) can interact with other terms in two ways: it can be a subterm, and its holes can hold subterms. So informally, we need to ensure LHS and RHS have same relation between "fringes", that is, holes and the topmost constructor. There is a rigorous proof of hygienic and some additional checks, which we omit here.

2.2 What they did wrong

There is a factual error: derivations of $\Sigma, C \vdash a \leqslant b$ might not be unique. In the following two examples, $x \leqslant y$ has a short derivation, indicated by the solid arrow, and a long one, indicated by the dotted arrow. We list corresponding scope rules beside. We state without proof that both set of rules are minimal, so there is no "smallest Σ" such that $\Sigma, C \vdash x \leqslant y$. The author might have missed the inference rule S-Lift in algorithmic rules, which is not syntax-directed.

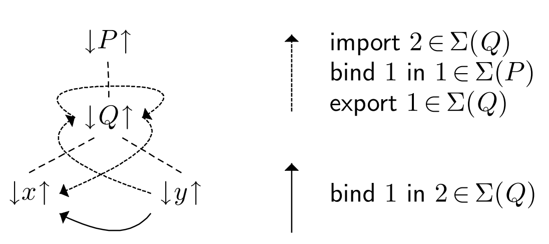


Figure 2. Why bind can go wrong.

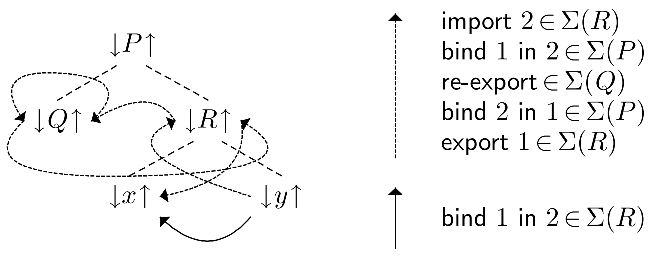

↓P↑
↓Q↑
↓R↑
↓x↑
↓y↑
import $2 \in \Sigma(R)$
bind 1 in $2 \in \Sigma(P)$
re-export $\in \Sigma(Q)$
bind 2 in $1 \in \Sigma(P)$
export $1 \in \Sigma(R)$
bind 1 in $2 \in \Sigma(R)$

Figure 3. Why re-export can go wrong

The error does not impact the correctness of the inference algorithm, but affect its "completeness". The algorithm neither compute the smallest set of rules, nor find a solution whenever there is one. We state without proof that, after removing re-export and bind i in i in their binding language, uniqueness of derivation holds.

Proposition 1. *If we remove* S-ReExport *and add $j \neq i$ in the premises of* SD-Bind *and* SA-Bind*, every derivation of $\Sigma, C \vdash a \leqslant b$, if any, is unique. Further, they are in the form* SA-Import*, …,* SA-Import*,* SA-Bind*,* SA-Export*, …,* SA-Export *interleaved with* S-Lift*. Graphically, the path consists of upward edges, followed by a cross edges, followed by downward edges.*

Proof. Use the algorithmic rules and their Theorem 4.3. Better, draw an example and it becomes clear. □

We also remove S-Decl, and change S-Var to

$$\frac{\Sigma, t \vdash {\downarrow} x^{\mathsf{R/D}} \leqslant {\uparrow} y^{\mathsf{D}}}{\Sigma, t \vdash x^{\mathsf{R/D}} \leqslant y^{\mathsf{D}}}.$$

We do not loose much expressivity due to the changes; none of their examples use those two rules.

Admittedly, we loose transitivity and preorder with this change. But it might be a good thing. Taking a close look, we find that the relation $\leqslant$ models two inherently different concepts: shadowing and reference. Sometimes we do not want $x^{\mathsf{R}} \leqslant x_2^{\mathsf{D}}$ even if $x^{\mathsf{R}} \leqslant x_1^{\mathsf{D}}$ and $x_1^{\mathsf{D}} \leqslant x_2^{\mathsf{D}}$. Consider a language with rich verification features. Specifically, you can declare ghost variables (or logic variables; you might refer to [MP96]), and interleave programs with assertions. You can refer to both ghost and program variable in assertions, so it makes sense for new program variables to shadow old ghost ones ($x_p^{\mathsf{D}} \leqslant y_g^{\mathsf{D}}$); but we do not want program statements to use ghost variables ($y^{\mathsf{R}} \leqslant x_p^{\mathsf{D}}, y^{\mathsf{R}} \nleqslant y_g^{\mathsf{D}}$). Transitivity would cause trouble here. Therefore, we leave transitivity broken, and the user should recover it herself.

Does that mean we also loose the correspondence to "binding as sets of scopes" and "scope as sets"? Observe that we can consider variables *with a same name* only to define bounded-ness. It seems taking a transitive closure to recover transitivity is harmless, as variables becoming visible are less specific anyway. The only exception is a symmetric pair, $x_1^{\mathsf{D}} \leqslant x_2^{\mathsf{D}}$ and $x_2^{\mathsf{D}} \leqslant x_1^{\mathsf{D}}$: we discover a new definition. We argue that such cycles are bad. In other words, we want a partial order.

- We can transform a preorder $\leqslant$ to a partial order $\preccurlyeq$ where $a \preccurlyeq b$ iff $a \leqslant b \wedge b \nleqslant a$. By definition, the transformation maintains the least elements of a set and thus the bound relation. This operation corresponds to creating scopes at binders instead of at binding forms in "binding as sets of scopes".

[2]. It is not so fair, since here we completely separate macro expansion and evaluation.

- Due to limitations of their binding language, cycles are necessary. In page 29 of the extended paper (link), we find that `Let` has discrete binders but `Letrec` has a big SCC. The difference is not intentional; it is a rescue, making every binder "equal" after creating "recursion". In our experience, cycles can mostly be avoided using our generalized binding language in Section 3.3. We also provide an algorithm to detect cycles in Section 4.2.

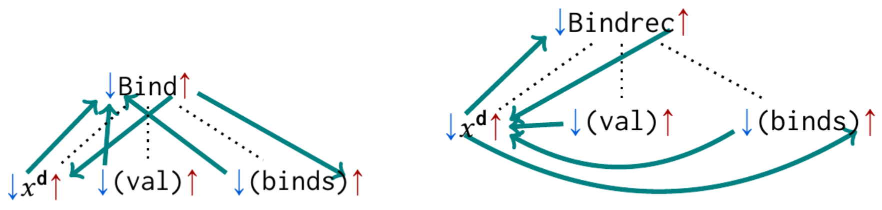


Figure 4. Scope rules for bindings of `Let` and `Letrec`

In the rest of the paper, we use → instead of ⩽. [hygiene?]

3 Inferring More Scope

We give a simple and elegant generalization to the work discussed in the previous section.

3.1 Motivation

The framework presented in [PKW17] is already expressive enough to model and infer almost all the binding forms in R^5RS, modulo the unsound `as_ref`. However, improvements are possible.

On page 23, they state that their "binding language can express this (`do`) scope", but their algorithm cannot "infer scope for it". But the "correct scope" they presented is wrong: we need `bind var in binds` so that `steps` inside `binds` can refer to earlier `vars`. This rule, however, would also enable `inits` to do so. So their binding language cannot express the scope of `do`. Intuitively, we cannot classify imports from `inits` and `steps` buried inside `binds`. Similar problem also appeared in `letrec`. As we discussed in Section 2.2, binding variables in `letrec` can refer to each other, inconsistent with that of `let`.

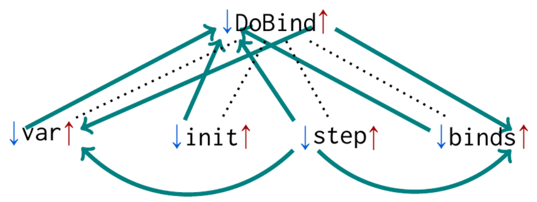


Figure 5. A wrong scope of `do`. On page 23 of [PKW17]

Another problem concerns the "untyped" nature of the language. Every constructor has one import port and one export port, but some of them are useless. If we add datatypes to the language, we will not use the export port of the `Expr` type, such as `lambda`, `app`, and so on. Why bother? If we write the macro

$$\texttt{let}(\texttt{end-binds}, e) \;\rightarrow\; e,$$

it would be reasonable to have the constraint

$$\textsf{export } 2 \in \Sigma(\texttt{let}) \;\Leftrightarrow\; \text{true},$$

which results in conflicting scope (although we could add that rule in advance, but that seems unnatural). To address the problem, they must wrap e in `begin` on page 4.

$$\texttt{let}(\texttt{end-binds}, e) \;\rightarrow\; \texttt{begin}(e),$$

If the core language does not offer a similar construct, we cannot write macros as simple as `let`.

3.2 Attempts

In this section, we discuss two given up attempt to address the `do` problem. As they suggested in the last paragraph, and inspired by Ott ([SNO+10]), our first attempt is to track two distinct set of scopes: one for `steps`, the other for others. I present the rules here.

Rule set 1 Σ_1(`DoBind`)	Rule set 2 Σ_2(`DoBind`)
import 1, 2	import 3, 4
	export 1, 4
	bind 1, 4 in 3
	bind 1 in 4

Unfortunately, doing so excludes some valid bindings. If there are nested `do` expressions as shown next, we need a combination of scopes, rather than mere union. Taking a transitive closure is enough, but as I said, we do not want transitivity built-in.

$$\mathtt{do}(\mathtt{x}, -, \mathtt{do}(-, \mathtt{var}(\mathtt{x}), -, -), -)$$

Multiple scopes are still useful in practice. For example, when modeling System F, we must separate variables and type variables.

The second attempt, as suggested in the last paragraph on page 23, is to extend the macro language with list patterns. We give a prototype definition here.

$$\begin{array}{rcl} l & \rightarrow & \bullet_l \\ & \mid & [] \\ & \mid & [t] \\ & \mid & l_1 + l_2 \\ t & \longrightarrow & \cdots \\ & \mid & P_l(l_1, \ldots, l_n) \end{array}$$

We introduce a new category of patterns matching lists. A list pattern can be a *list* hole, an empty list, a singleton of normal pattern, or concatenation of two lists. A normal pattern can be a *list* constructor: just like ellipsis in `syntax-rules`, it matches a list of productions, and returns a production of lists. For example,

$$\mathsf{match}([\mathtt{P}(\mathtt{x},\mathtt{y}), \mathtt{P}(\mathtt{z},\mathtt{w})], \mathtt{P}(\bullet_1, \bullet_2)) \;=\; \{\bullet_1 \mapsto [\mathtt{x},\mathtt{z}], \bullet_2 \mapsto [\mathtt{y},\mathtt{w}]\}.$$

On the right hand side, list constructors do the reverse. We treat a list as a whole when it comes to scope rules: there are no rules inside elements of a list; if P_l imports a list, then it imports all elements in the list. The overall effect is roughly equivalent to adding a pre- and post-processing phase before and after macro expansion, respectively, and use familiar `cons` and `nil` with suitable scope rules to store lists.

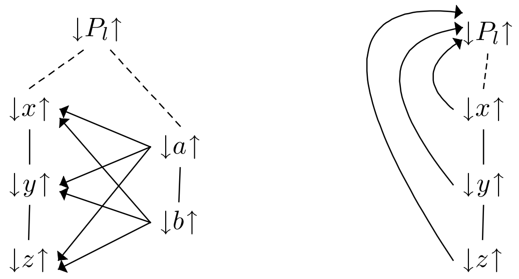


Figure 6. How scope rules control lists. The right shows import; the left shows bind

Although list patterns do solve the problem at hand, it is quite ad-hoc. First, comparing to ellipsis, we are only allowed to manipulate nested lists in a restrictive way. We can transform the outermost list freely, but we cannot transform inner lists as well, as shown next.

```
(syntax-rules (P)
  [((P x) ...)
    ((Q y x) ...)])
```

$$\mathtt{P}^l(x) \;\rightarrow\; \mathtt{Q}^l(\mathtt{ys}, x)$$

```
(syntax-rules (P Q)
  [((P ((Q x) ...)) ...)
    ((P ((R y x) ...)) ...)])
```

Figure 7. The macro in the middle expresses the Scheme macro on the left, but the one on the right does not omit an equivalent definition

Second, if the shape of the expression is not a list at all (as we shall see later), we are doomed. Despite the limitations, list patterns are convenient for developers on its own.

3.3 Solution

We solve the problems by using a variable number of ports. Unlike Ott, we allow ports to mix with each other; in fact, we work in a many-sorted algebra setting. There are two primitive sorts: `VarDef` with one import port and one export port, and `VarUse` with only one import port. Now we show how the extended binding language can express the scope of `do`, using two import ports. For simplicity, we omit the body of `do`, leaving only a series of bindings. Note that it is not the minimum set of rules, only a minimal set.

Expr: 1 import
Stx: 2 import, 1 export

DoBind: VarDef → Expr → Expr → Stx → Stx	Do: Stx → Expr
2 import $1, 2, 4.2 \in \Sigma(\mathtt{DoBind})$	import $1.1, 1.2 \in \Sigma(\mathtt{Do})$
1 import $3, 4.1 \in \Sigma(\mathtt{DoBind})$	
export $1, 4 \in \Sigma(\mathtt{DoBind})$	
bind $1, 4$ in $3 \in \Sigma(\mathtt{DoBind})$	
bind 1 in $3.1 \in \Sigma(\mathtt{DoBind})$	

Table 1. `do`'s constructs. We do not explain the notations here, but it should be clear

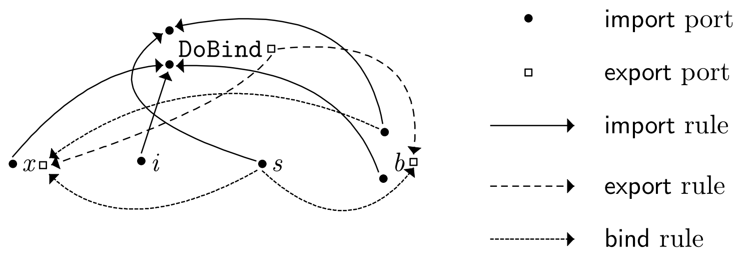


Figure 8. Graphical illustration of `do`'s scope rules. Rules of `Do` are omitted

Of course, we also need to show that we can infer the scope from syntactic macros. The inference algorithm is essentially the same: we enumerate pairs of ports of the holes, and impose an equivalent relation on the RHS and LHS of the macro. But we need a full-fledged SAT solver to solve the constraints, which is in general not in polynomial time. Before we give our inference algorithm, let us see what the macros look like.

$$\begin{aligned}
\texttt{Do}(\bullet) &\rightarrow \texttt{extracting}(\bullet, \texttt{vnil}, \texttt{enil}, \texttt{enil})\\
\texttt{extracting}(\texttt{DoBind}(\bullet_x, \bullet_i, \bullet_s, \bullet_b), \bullet_{xs}, \bullet_{is}, \bullet_{ss}) &\rightarrow \texttt{extracting}(\bullet_b, \texttt{vcons}(\bullet_x, \bullet_{xs}), \texttt{econs}(\bullet_i, \bullet_{is}), \texttt{econs}(\bullet_s, \bullet_{ss}))\\
\texttt{extracting}(\texttt{EndDoBind}, \bullet_{xs}, \bullet_{is}, \bullet_{ss}) &\rightarrow \texttt{letrec}(f, \texttt{lambda}(\bullet_{xs}, \texttt{app}(f, \bullet_{ss})), \texttt{app}(f, \bullet_{is}))
\end{aligned}$$

The core-language constructs and the auxiliary function `extracting` are listed in the following. You can check the "if and only if" condition, though a little tedious.

`VList`: 1 import, 1 export
`EList`: 1 import

`extracting`: `Stx` → `VList` → `EList` → `EList` → `Expr`
import $1.1, 1.2, 2, 3, 4 \in \Sigma(\texttt{extracting})$
bind 2 in $1, 4 \in \Sigma(\texttt{extracting})$
bind 1.1 in $4 \in \Sigma(\texttt{extracting})$

`letrec`: `VarDef` → `Expr` → `Expr` → `Expr`
import $1, 2, 3 \in \Sigma(\texttt{letrec})$
bind 1 in $2, 3 \in \Sigma(\texttt{letrec})$

`vcons`: `VarDef` → `VList` → `VList`
import $1, 2 \in \Sigma(\texttt{vcons})$
export $1, 2 \in \Sigma(\texttt{vcons})$

`econs`: `Expr` → `EList` → `EList`
import $1, 2 \in \Sigma(\texttt{econs})$

`app`: `Expr` → `EList` → `Expr`
import $1, 2 \in \Sigma(\texttt{app})$

`lambda`: `VList` → `Expr` → `Expr`
import $1, 2 \in \Sigma(\texttt{lambda})$
bind 1 in $2 \in \Sigma(\texttt{lambda})$

We present the algorithm here, which carefully generates a polynomial number of clauses. Sanity checks, and constraints for discarded holes and fresh variables, are omitted.

Algorithm 1

```
function path($e_l, p_l, e_r, p_r$):
    if $(e_l, p_l, e_r, p_r) \in \mathrm{dom}(memo)$
        return $memo(e_l, p_l, e_r, p_r)$
    if is_import($p_l$) $\wedge$ is_import($p_r$)
        if mother($e_l$) $\neq e_r$
            $v :=$ new_var(), $w := \{\}$, $e_m :=$ mother($e_l$)
            for $p_m \in$ imports(function($e_m$))
                $w := w \cup \{$path($e_m, p_m, e_r, p_r$) $\wedge$ import(function($e_m$), $p_m$, child_number($e_l$), $p_l$)$\}$
            add_clause($v \leftrightarrow \bigvee w$)
            $memo(e_l, p_l, e_r, p_r) := v$
            return $v$
        else return import(function($e_r$), $p_r$, child_number($e_l$), $p_l$)
    elseif is_export($p_l$) $\wedge$ is_export($p_r$)
        if mother($e_r$) $\neq e_l$
            $v :=$ new_var(), $w := \{\}$, $e_m :=$ mother($e_r$)
            for $p_m \in$ exports(function($e_m$))
                $w := w \cup \{$path($e_l, p_l, e_m, p_m$) $\wedge$ export(function($e_m$), $p_m$, child_number($e_r$), $p_r$)$\}$
            add_clause($v \leftrightarrow \bigvee w$)
            $memo(e_l, p_l, e_r, p_r) := v$
            return $v$
        else return export(function($e_l$), $p_l$, child_number($e_r$), $p_r$)
    elseif is_import($p_l$) $\wedge$ is_export($p_r$)
        if mother($e_l$) = mother($e_r$)
            return bind(mother($e_l$), child_number($e_l$), $p_l$, child_number($e_r$), $p_r$)
        else omitted...

function generate_constraint($c_l, c_r$):
    if is_function($c_r$)
        for $i \in \mathbb{N}$, find($\bullet_i, c_l$) = $h_l$, find($\bullet_i, c_r$) = $h_r$
            for $p_1 \in$ imports(function($h_l$)), $p_2 \in$ imports(function($c_l$))
                add_clause(path($h_l, p_1, c_l, p_2$), path($h_r, p_1, c_r, p_2$))
            for $p_1 \in$ exports(function($h_l$)), $p_2 \in$ exports(function($c_l$))
                add_clause(path($c_l, p_2, h_l, p_1$) $\leftrightarrow$ path($c_r, p_2, h_r, p_1$))
        for $i, j \in \mathbb{N}, i \neq j$, find($\bullet_i, c_l$) = $h_l$, find($\bullet_i, c_r$) = $h_r$, find($\bullet_j, c_l$) = $h'_l$, find($\bullet_j, c_r$) = $h'_r$
            for $p_1 \in$ imports(function($h_l$)), $p_2 \in$ imports(function($h'_l$))
                add_clause(path($h_l, p_1, h'_l, p_2$) $\leftrightarrow$ path($h_r, p_1, h'_r, p_2$))
    else omitted...
```

Remark 2. There may exist other generalizations. If we stick to the unique-derivation-rules framework, then we can generalize "arrows" to have any attribute that forms a monoid equipped with a homomorphism to $(\{\bot, \top\}, \wedge)$. If we work in a many-sorted setting, we can generalize to categories. Inference then means solving equations

$$\begin{array}{rcl} x_1 \cdots x_n & = & x'_1 \cdots x'_m \\ & \vdots & \end{array}$$

This problem is known as the *unification problem* or the *Diophantine problem*, and is generally undecidable. Those monoids that the unification problem is solvable (free monoids, see [Mak77]) do not seem to have clear meaning in terms of scopes. Here we choose the category **FinRel** and require the user to assign objects (i.e., number of ports) to sorts.

In the rest of the paper, we will use the following notations. $P\colon (x\colon A) \to (y\colon B) \to C$ means P is a function of sort C with parameters x, y with sorts A, B respectively. $x.p{\uparrow}q \in P$ means the port q of P imports the port p of parameter x. Similarly, we use notations $p{\downarrow}x.q \in P$, and $x.p \to y.q \in P$ for exports and binds, respectively. Sometimes we use natural numbers to name ports, sometimes we use pretty names. If there is a single port, we sometimes omit the number.

3.4 Case studies

In this section, we shall use a primitive multi-arm let. Here we give its scope rules; other core constructs are standard (at least for a Scheme programmer), simply importing every child.

$$\begin{array}{rcl}
\mathsf{Expr} & :: & 1\ \mathsf{import}, 0\ \mathsf{export} \\
\mathsf{Bind} & :: & 1\ \mathsf{import}, 1\ \mathsf{export} \\
\mathtt{LetBind} & : & (x\colon \mathtt{VarDef}) \to (e\colon \mathtt{Expr}) \to (b\colon \mathtt{Bind}) \to \mathtt{Bind} \\
x{\uparrow}, e{\uparrow}, b{\uparrow}, x{\downarrow}, b{\downarrow} & \in & \mathtt{LetBind} \\
\mathtt{let} & : & (b\colon \mathtt{Bind}) \to (e\colon \mathtt{Expr}) \to \mathtt{Expr} \\
b{\uparrow}, e{\uparrow}, e \to b & \in & \mathtt{let}
\end{array}$$

As already mentioned, the situation of `letrec` is similar to that of `do`. We present `letrec`'s scope rules and macros here, but omit the scope rules of the auxiliary function. Note that the style of the scope rules is a little different from `do`.

$$\begin{array}{rcl}
\mathtt{Stx} & :: & 2\ \mathsf{import}, 1\ \mathsf{export} \\
\mathtt{RecBind} & : & (x\colon \mathtt{VarDef}) \to (e\colon \mathtt{Expr}) \to (b\colon \mathtt{Stx}) \to \mathtt{Stx} \\
\begin{array}{l} e{\uparrow}0, b.0{\uparrow}0, x{\uparrow}1, e{\uparrow}1, b.1{\uparrow}1, x{\downarrow}, b{\downarrow}, \\ e \to x, e \to b, b.0 \to x \end{array} & \in & \mathtt{RecBind} \\
\mathtt{letrec} & : & (b\colon \mathtt{Stx}) \to (e\colon \mathtt{Expr}) \to \mathtt{Expr} \\
b.1{\uparrow}, e{\uparrow}, e \to b.0 & \in & \mathtt{letrec}
\end{array}$$

$$\begin{array}{rcl}
\mathtt{letrec}(\bullet_b, \bullet_e) & \to & \mathtt{aux}(\bullet_b, \mathtt{EndLetBind}, \mathtt{void}, \bullet_e) \\
\mathtt{aux}(\mathtt{RecBind}(\bullet_x, \bullet_e, \bullet_b), \bullet_{bs}, \bullet_{ss}, \bullet_{body}) & \to & \mathtt{aux}(\bullet_b, \mathtt{LetBind}(\bullet_x, \mathtt{void}, \bullet_{bs}), \mathtt{begin}(\mathtt{set!}(\mathit{def}{\to}\,\mathit{ref}(\bullet_x), \bullet_e), \bullet_{ss}), \bullet_{body}) \\
\mathtt{aux}(\mathtt{EndRecBind}, \bullet_{bs}, \bullet_{ss}, \bullet_{body}) & \to & \mathtt{let}(\bullet_{bs}, \mathtt{begin}(\bullet_{ss}, \bullet_{body}))
\end{array}$$

There is an undocumented but necessary feature here, which already exists in their implementation: generating a variable reference from a variable definition, or `as_refn$x`. It is used in `letrec`, named `let`, and so on. The problem is that they only check that `as_refn$x` is indeed bound by `x`. They do not ensure that `as_refn$x` is uniquely bound, nor that `x` is not shadowed by some other definitions. Unrestricted use of `as_refn` can therefore break hygiene. There are some criteria (the proof is left as an exercise) that implies a export port of a hole • do not contain a unwanted definition `x'`, such as (1) `as_refn$x` does not import that port; (2) `x` imports that port; and (3) no import port of • can reach `x`, and in LHS something imports both `x` and that port. However, in practice, these requirements are too strong. Instead of imposing constraints *a priori*, we choose to verify hygiene after inference *a posteriori*. The downside is that we have to recheck every use of `as_refn` after each time we extend the language. We will explain how we check this property, along with others, in Section 4.2.

The second example, `match`, shows why more than one export port is useful. We only show the scope rules here, because the macros are too complex (albeit straightforward). You can find them in the source code repository.

$$\begin{array}{rcl}
\texttt{Pattern} & :: & 2\ \mathsf{import}, 2\ \mathsf{export} \\
\texttt{var-pat} & : & (x\colon \texttt{VarDef}) \to \texttt{Pattern} \\
x{\uparrow}1, x{\downarrow}0, x{\downarrow}1 & \in & \texttt{var-pat} \\
\texttt{pair-pat} & : & (p_1\colon \texttt{Pattern}) \to (p_2\colon \texttt{Pattern}) \to \texttt{Pattern} \\
\begin{array}{r} p_1.0{\uparrow}0, p_2.0{\uparrow}0, p_1.1{\uparrow}1, p_2.1{\uparrow}1, \\ p_1.0{\downarrow}0, p_2.0{\downarrow}0, p_1.1{\downarrow}1, p_2.1{\downarrow}1, \\ p_1.0 \to p_2.1, p_2.0 \to p_1.1 \end{array} & \in & \texttt{pair-pat} \\
\texttt{aut-pat} & : & (f\colon \texttt{Expr}) \to (x\colon \texttt{VarDef}) \to \texttt{Pattern} \\
f{\uparrow}0, x{\uparrow}0, x{\downarrow}0 & \in & \texttt{aut-pat} \\
\texttt{match} & : & (e\colon \texttt{Expr}) \to (p\colon \texttt{Pattern}) \to (g\colon \texttt{Expr}) \to (r\colon \texttt{Expr}) \to \texttt{Expr} \\
e{\uparrow}, p.0{\uparrow}, p.1{\uparrow}, g{\uparrow}, r{\uparrow}, g \to p.1, r \to p.0 & \in & \texttt{match}
\end{array}$$

Here we try to model "Indiana's match" (unofficial link; something similar in the nanopass framework [KD13]), where you can apply a function on a matched item immediately (called an "automorphism"). The body can use the matched-then-transformed variables, but the guard cannot, as the latter is evaluated before the transformations. Thus we need to classify two different sets of bindinds buried in the pattern, a job calling for two ports.

4 Computability and Complexity

4.1 NP completeness

In the previous section, we use a SAT solver to infer scope (given a fixed number of ports). We would wonder whether there are more efficient algorithms. We show that this problem is NP-hard, and thus NP-complete; thus a EXP-time algorithm is the best we can hope for, at least for now.

We reduce the 3-CNF satisfiability problem, SAT, to our scope inference problem, SCOPE. We first give a reduction using an unlimited number of ports. Consider the formula $\psi = \varphi_1 \wedge \cdots \wedge \varphi_m$ with n variables and m clauses. We list core-language constructs in the following table.

Sort	Function	Scope rule
L with $2n+1$ import ports	$X\colon (v\colon L) \to A$	$v.x{\uparrow}0 \in X$
A with 1 import ports	$Y\colon (a\colon A) \to L$	$a.0{\uparrow}x \in Y$
	$I\colon (a\colon A) \to A$	$a.0{\uparrow}0 \in I$
	$N\colon (a\colon A) \to A$	(no rule for N)
	$D_j\colon (l\colon L) \to A$	$l.x_j{\uparrow}0 \in D_j$
	$\bar{D}_j\colon (l\colon L) \to A$	$l.\bar{x_j}{\uparrow}0 \in \bar{D}_j$
	$T_j\colon (a\colon A) \to L$	$a.0{\uparrow}x_j \in T_j$
	$\bar{T}_j\colon (a\colon A) \to L$	$a.0{\uparrow}\bar{x}_j \in \bar{T}_j$
	$C_i\colon (a\colon A) \to A, 1 \leqslant i \leqslant m$	$a.0{\uparrow}l \in C_i$ iff $l \in \varphi_i$

Here we name the ports of L as $x_1, \ldots, x_n, \bar{x_1}, \ldots, \overline{x_n}, v$ for clarity. Intuitively, C_i characterizes φ_i. The use of each functions will be clear soon.

The inference problem consists of a new function $V\colon (c\colon L) \to L$ and 3 classes of macros. The first class characterizes ψ: for each $1 \leqslant i \leqslant m$, construct the macro

$$X(V(C_i(\bullet))) \quad \to \quad I(\bullet).$$

Intuitively, we want $c.l{\uparrow}v \in V$ iff $l = 1$. Here is an example for $\varphi = a \vee \bar{b}$.

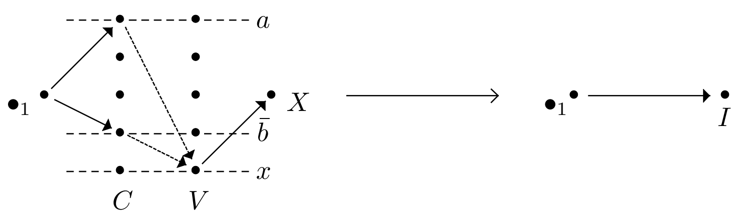


Figure 9. The macro corresponding to $\varphi = a \vee \bar{b}$. At least one of the dotted lines should hold

The second class ensure that $c.l{\uparrow}v \in V$ iff $c.v{\uparrow}l \in V$. For each $1 \leqslant j \leqslant n$, construct the macro

$$X(V(T_j(\bullet))) \quad \to \quad D_j(V(Y(\bullet))),$$

and similarly for $\bar{T}_j$ and $\bar{D}_j$. Here is an example.

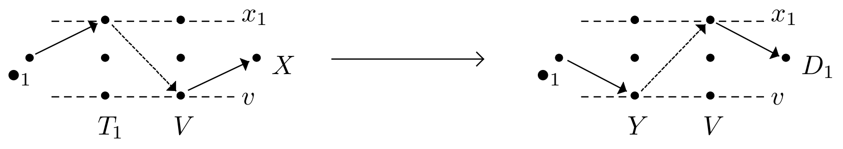


Figure 10. The macro for x_1. The dotted lines should hold at the same time

The third class of macros ensure that $c.l\uparrow v \in V$ and $c.\bar{l}\uparrow v$ are exclusive. For each $1 \leqslant j \leqslant n$, construct the macro

$$D_j(V(V(\bar{T}_j(\bullet)))) \rightarrow N(\bullet).$$

Here is a pictural demonstration. Notice that thanks to previous macros, the dotted (dashed) lines must hold at the same time.

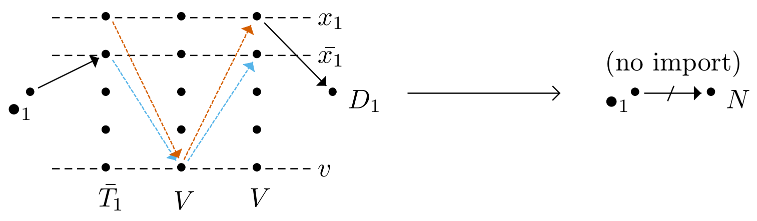


Figure 11. The macro for x_1 and $\bar{x}_1$. The blue and red lines cannot coexist

We do not prove it formally, but it should be clear now that those macros characterize the satisfiability problem for ψ precisely. Also, there are only a polynomial number $(m + 3n)$ of macros.

Proposition 3. *The macros defined such can be given a non-conflicting scope with respect to the core language if and only if ψ is satisfiable. Further, this procedure defines a Karp reduction from* SAT *to* SCOPE.

Next we give a reduction using only a fixed number (4) of ports. Whether less ports suffice is left as an open problem. Again, we present the core-language in the following tabe.

Sort	Function	Scope rule
B with 4 import ports	$T: (v: B) \rightarrow B$	$v.d\uparrow d \in T, v.d\uparrow t \in T$
U with 1 import ports	$F: (v: B) \rightarrow B$	$v.d\uparrow d \in F, v.d\uparrow f \in F$
	$S: (u: U) \rightarrow B$	$u.0\uparrow d \in S$
	$E: (b: B) \rightarrow U$	$b.w\uparrow 0 \in E$
	$I: (u: U) \rightarrow U$	$u.0\uparrow 0 \in I$

Here we name the ports of B as d, t, f, w. The inference problem consists of new functions $X_i: (c: B) \rightarrow B$ for each variable, and macro

$$E(X_i(T(X_j(F(X_k(T(S(\bullet)))))))) \rightarrow I(\bullet)$$

for each clause $x_i \vee \bar{x_j} \vee x_k$ (generalized to other combination of signs). Here is an example to model the clause $a \vee \bar{b}$.

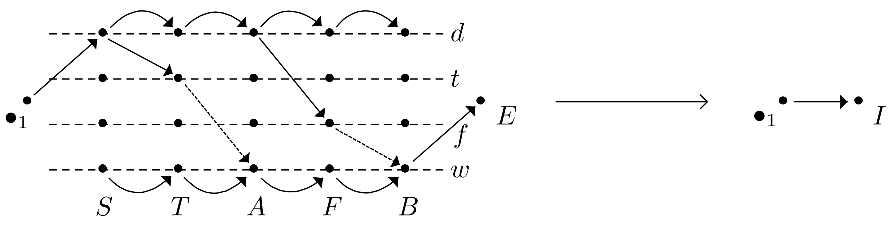


Figure 12. The macro corresponding to $a \vee \bar{b}$. At least one dotted line should hold

Intuitively, we want $c.t\uparrow w \in X_i$ iff $x_i = 1$, and $c.f\uparrow w \in X_i$ iff $x_i = 0$. We use the same trick in the first reduction to ensure they are exclusive, which we omit here.

Another variant of SCOPE, where we cannot import a port twice, is also NP-complete. We will not elaborate here, just point out that we can "ban" imports (using macros) instead of "postulate" allowed imports of T and F in the second reduction. We only need one of them, after all.

4.2 Scope as regular languages

We give a trivial but somewhat surprising characterization of the expressivity of our binding language. Let us first talk about the expressivity of their binding language.

Although they stated in Section 3.1 that scoping rules are local, the resulting scoping might not be so local (but still lexical). In Figure 2, we cannot tell whether $y \rightarrow x$ from the subtree rooted in Q, because of the possibility of bind 1 in $1 \in \Sigma[P]$. Similarly, in Figure 3, we cannot tell whether $y \rightarrow x$ from the subtree rooted in R. It is debatable whether this feature is good or bad, but it is indeed a little peculiar.

Now that we obtained uniqueness of derivations by removing those two kinds of rules, a natural question is how we can tell a scoping (instead of a single reference-definition pair) can be expressed by our binding language or not. It becomes more important in the presence of the "variadic port" extension, because we have potentially infinite candidates to try with. It turns out that we can express a reference relation (resp. a shadowing relation) if and only if it is, in some sense, a regular language. It sounds like a reminiscence of [NTVW15], where they use "labeled import" and regular expressions to customize reachability and visibility relations. We first introduce a variant of our binding language, using one kind of ports and no `bind` rules.

Definition 4. *Each sort has a finite number of ports. Same as our binding language,* `VarUse` *has one port and* `VarDef` *has two ports.*

Each function specify directed edges between its ports and its arguments' ports. We write $p.a$ `up` *$b \in f$ if there is an edge from the port a of argument p to the port b of function f; we define similarly for* `downs`*.*

A reference x can refer to a definition y, denoted $x \rightsquigarrow y$, iff there is a path of ports from x to y over the simple path of tree nodes *on the tree; we define similarly for shadowings.*

Proposition 5. *For each instance in the language of the previous definition, there is an instance in our binding language, such that $x \rightsquigarrow y$ iff $x \rightarrow y$.*

Proof. $\Rightarrow$: We duplicate each port. One of them serve for `ups`, now `imports`, only, and the other for `downs` or `exports`. Then we "inline" `ups` followed by `downs` to `binds`. Here is what happens.

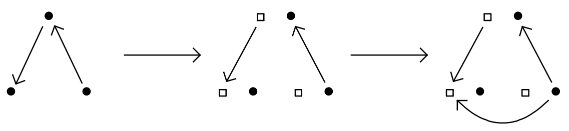


Figure 13. How we transform rules for each function, from $\rightsquigarrow$ to $\rightarrow$.

$\Leftarrow$: We create a intermediate port for each `bind`. Here is what happens.

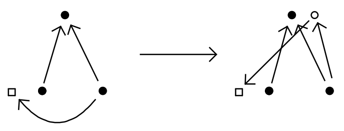


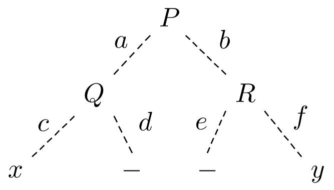
Figure 14. How we transform rules for each function, from $\rightarrow$ to $\rightsquigarrow$.

$\square$

Remark 6. There are two reasons why we do not remove the "simple path" restriction and use the more powerful language. First, we can generally not describe *paths* exactly. For example, the language of well-sorted paths is not regular. Second, We have to modify the number of ports. For example, two ports for `Expr` is enough for λ but not for `do`.

The language in Definition 4 reminds us of automata and states. We formalize what we think.

Definition 7. *We describe a simple path between two leaves in an AST using a list of $f.p^{\uparrow/\downarrow}$, where f is a function and p is one of its parameter. The arrow indicates the direction of an edge. For example, in the following tree, the simple path from x to y is described by $Q.c^{\uparrow}, P.a^{\uparrow}, P.b^{\downarrow}, R.f^{\downarrow}$.*



We sometimes abbreviate (a segment of) the first half of a path, ending at the least common ancestor, as $p^{\uparrow}$. Similarly for $p^{\downarrow}$. We use yet another arrow $\hookrightarrow$ to denote the path between two leaves.

Proposition 8. *For each instance in the language of Definition 4, there are two regular languages L_1, L_2 in the alphabet $\{f.p^{\downarrow/\uparrow}\}$ such that $x^{\mathsf{R}} \rightsquigarrow y^{\mathsf{D}}$ iff $x^{\mathsf{R}} \hookrightarrow y^{\mathsf{D}} \in L_1$, $x^{\mathsf{D}} \rightsquigarrow y^{\mathsf{D}}$ iff $x^{\mathsf{D}} \hookrightarrow y^{\mathsf{D}} \in L_2$.*

Proof. $\Rightarrow$: We construct an NFA for L_1, and similarly for L_2. The states are $\biguplus \{ports(s) \mid s \in sorts\}$. There is a transition $s \xrightarrow{f.p^{\uparrow}} t$ for each rule $p.t$ `up` $s \in f$, and similarly for `downs`. The initial state is the unique port of `VarUse`, and the final state is the `export` port of `VarDef`. The equivalence holds by construction. Notice that L_1 here only includes valid well-sorted paths. In Proposition 8 we do not require this, because we can take the intersection with the language of well-sorted path anyway.

$\Leftarrow$: Without loss of generality, suppose we are given a NFA $(S, \rightarrow)$ for L_1 with a single initial state and a single final state. We assign each sort $|S|$ states. There is a rule $p.t$ `up` $s \in f$ for each transition $s \xrightarrow{f.p^{\uparrow}} t$, ans similarly for `downs`. We assign a disjoint set of ports for another NFA for L_2 and construct rules similarly. The equivalence holds by construction. $\square$

Corollary 9. *For each instance in our binding language, there are two regular languages L_1, L_2 such that $x^{\mathsf{R}} \rightarrow y^{\mathsf{D}}$ iff $x^{\mathsf{R}} \hookrightarrow y^{\mathsf{D}} \in L_1$, $x^{\mathsf{D}} \rightarrow y^{\mathsf{D}}$ iff $x^{\mathsf{D}} \hookrightarrow y^{\mathsf{D}} \in L_2$.*

Remark 10. We can also describe the fixed version in Section 2 as language recognized by a 1-state DFA on the alphabet, say, $\{x^{\uparrow}, x^{\downarrow}, x^{\rightarrow}\}$. But this is fragile: we must use the extended alphabet, and we cannot remove the intersection. On the other hand, our binding language is more robust. For example, we can use different alphabets, we can "reverse" the second half of the path and still get an regular language, and we can impose the intersection condition or not.

We can state a "precise" version of the inference problem. The inference algorithm Algorithm 1 requires the developer to instruct the number of ports, and fix them forever. Although it makes sense in practice, the algorithm might fail on some situations where the intended scope is expressible itself. The precise problem can be stated in terms of formal languages theory as follows.

Definition 11. *We define* inference problem *to mean: given a regular language $L \subseteq \Sigma^*$ and a finitely presented Thue system R in a finitely extended alphabet $\Sigma_1 \supseteq \Sigma$, decide whether there is a regular language $L_1 \subseteq \Sigma_1^*$, such that L_1 is closed under R, and $L_1 \cap \Sigma^* = L$.*

This decision problem seems hard, but proving it is undecidable is also hard. I posted this problem online (link); it received some attention, but no one has posted an answer yet.

Conjecture 12. *The inference problem is undecidable. Further, it is also undecidable if only newly introduced sorts have unknown ports.*

Now we give several applications of the "regular language" backend. We call the languages $x^{\mathsf{R}} \xrightarrow{p} y^{\mathsf{D}}$ as R, and $x^{\mathsf{D}} \xrightarrow{p} y^{\mathsf{D}}$ as D.

First, we can check whether two set of scope rules describe the same scope, because language equivalence for regular languages is decidable. It is useful if the inference algorithm reports several solutions, and we want to see if all of them are equivalent (although, since we use explicit ports, they might still behave differently in subsequent extensions).

Second, we can compute the transitive version of a set of scope rules as simple as $\hat{R} = RD^*, \hat{D} = D^*$. Note that they are no longer of the form $p^{\uparrow}q^{\downarrow}$; it is reasonable, because the intermediate x^{D} must exist for transitivity to work.

It is a litter more complex to check whether a given set of scope rules is transitive *per se*. We take $\hat{R}$ and $\hat{D}$ defined previously, and extend them with the rewriting rule $x^{\uparrow}x^{\downarrow} \to \varepsilon$. Then we intersect them with well-sorted simple paths. This way, we get all "candidate" paths of the transitive version, R' and D'. Then we check whether $R = R'$ and $D = D'$. Why we can extend the language is omitted; please see [Leu08].

Third, we can check whether a set of scope rules is free of cyclic shadowing relation. We reverse the language D and flip the alphabet ($x^{\uparrow}$ to $x^{\downarrow}$ and vice versa) to get the "true" reverse $\tilde{D}$. Then we check whether $D \cap \tilde{D} = \varnothing$. We can also check whether a hole $\bullet$ can shadow `x` in `as_refn$x`. Let the path from `as_refn$x` to $\bullet$ be p, and the path from $\bullet$ to `x` be q, we should check whether

$$\exists r, pr^{\downarrow} \in R \wedge r^{\uparrow} q \in D.$$

Note that a regular language L prefixed (resp. postfixed) by a certain string s, $\mathit{pre}(L, s)$ (resp. $\mathit{post}(L, s)$), is again a regular language. So it reduces to checking emptiness of $\mathit{pre}(R, p) \cap \mathit{flip}(\mathit{post}(D, q))$.

Fourth, we get a name resolution algorithm for a single reference for free. We traverse the AST from the reference and follow the DFA of R. In case multiple definitions are accepted, we further traverse from each one and follow D to decide the shadowing relation. This algorithm works in constant space and $O((1 + c)\,n)$ time, where c is the number of found definitions and n is the size of the tree. This is quite long for a single query, but notice that different searches from each definition and different branches of a search can be executed in parallel.

There is a simple optimization for this algorithm. In many cases, a "close" definition will shadow other possible ones. Once we find such a definition, we can stop early. We consider a simpler case: in a cyclic-shadowing–free scope, if $x^{\mathsf{R}} \xrightarrow{p^{\uparrow}q^{\downarrow}} x^{\mathsf{D}}$ and for all y and $r = x^{\uparrow}r' \in \mathit{WS}$, $x^{\mathsf{R}} \xrightarrow{p^{\uparrow}r} x^{\mathsf{D}}$ implies $x^{\mathsf{D}} \xrightarrow{q^{\uparrow}r} y^{\mathsf{D}}$, then we can stop at the subtree (why?). We note that there are finitely many $\mathit{pre}(L, s)$ for a fixed L, and

$$\mathit{cong}(L, s_0) \;\triangleq\; \{s \mid \mathit{pre}(L, s) = \mathit{pre}(L, s_0)\}$$

is regular (think about finite automata). We enumerate those languages and decide the inclusion

$$\mathit{safe}(p^{\uparrow}, q^{\uparrow}) \;\triangleq\; \mathit{pre}(R, p^{\uparrow}) \cap \mathit{WS} \;\subseteq\; \mathit{pre}(D, q^{\uparrow}) \cap \mathit{WS}.$$

Here WS is the regular language of well-sorted simple paths starting with an upward arrow. Then we get the final language for "strong" reference $p^{\uparrow}q^{\downarrow}$ by

$$R' \;\triangleq\; R \cap \bigcup \{\mathit{cong}(R, p^{\uparrow}) \circ \mathit{flip}(\mathit{cong}(D, q^{\uparrow})) \mid \mathit{safe}(p^{\uparrow}, q^{\uparrow})\}.$$

We do not cover every possible usage here, such as in finding references of a given definition.

5 Implementation and Evaluation

5.1 Osazone integration

I implemented the fixed version in Section 2 in Osazone, with additional support for multiple scopes. The developer should mark datatypes with attribute in `Lang.osa` indicating which scopes are needed, for example,

```
[#scope Exp Type]
data Constr
  = ConstrCons Id TypeList Constr
  | ConstrNil
```

The developer specify scope rules in a separate file, say `scope.scp`, the scope rules. For example,

```
rules Exp
  Lang.ConstrCons :: import 1 3
                     export 1 3
                     def 1
  Lang.EVar  :: import all
  Lang.EAbs  :: import 1 3
                bind 1 in 3
                def 1
  Lang.ELet  :: import all
                bind 1 in 3
                def 1
```

Here `def` marks a definition, because in existing languages we use type `Id` in both definitions and references. As just illustrated, we support abbreviations such as `import all`. The language generator then, among other things, generates code that scope check a whole program in `Lib/Scope.hs`. The code is quite straightforward, merging balanced binary trees (`Maps`) everywhere, so you should not expect it to scale up.

In a purely functional setting, we use lists of indexes to locate in AST.

```
type Location = [Int]
type LocationTable = Map.Map Identifier.Id [Location]
```

We resolve names in two passes, which comes as no suprise if we think in attribute grammars ([Knu68]). For each scope, we create a typeclass

```
class ScopeableExp a where
  collectExportExp :: Location -> a -> ExportedId
  collectImportExp :: LocationTable -> [ExportedId] -> a -> References
```

Here information are stored in tree-shaped data structures.

```
data ExportedId = IC LocationTable [ExportedId]
data References
  = RefNode [References]
  | RefVar  [Location]
  | RefOmit
```

The generated code looks like this.

```
collectExportExp p (Lang.EAbs e1 _ e3) =
  let ic1 = collectExportExp (1 : p) e1
      ic3 = collectExportExp (3 : p) e3
  in IC (Map.unionsWith (++) []) [ic1, ic3]
collectImportExp m [IC m1 cs1, IC m3 cs3] (Lang.EAbs e1 _ e3) =
  RefNode [ collectImportExp (Map.unionWith (++) m $ Map.unionsWith (++) [m1]) cs1 e1,
            RefOmit,
            collectImportExp (Map.unionWith (++) m $ Map.unionsWith (++) [m1]) cs3 e3 ]
```

The inference algorithm is standard. I also implemented API to use scope checking facilitied in Osazone from Emacs. The developer should craft a Tree-sitter parser and register it in Emacs. She also needs to write, mechanically, eLisp code to serialize the syntax tree to S-expressions.[3] Our eLisp program opens a inferior Osazone process. Each time the programmer execute the `find-definition-at-point` command, our program sends the S-expression of the current syntax tree and the queried variable's location to Osazone, who responds with error or location of the definition. Emacs then moves the cursor to corresponding location in the buffer, using information in the syntax tree.

5.2 Standalone implementation

I implemented the generalized version in Section 3.3 in C++, with the help of MiniSat (see, for example, [ES04]). The source code is maintained at Github (link). You can write declaratively in the following syntax:

program	→	*stmt*+
stmt	→	`(defsort` *sid* `:import` *num* `:export` *num*`)`
	|	`(defun` *fid* `(:` *pid sid*`)`∗ `:sort` *sid*`)`
	|	`(defscope` *fid* `:import (`*port*∗`)`∗ `:export (`*port*∗`)`∗ `:bind (`*port port*`)`∗`)`
	|	`(defmacro` *expr expr*`)`
	|	`(regexp` *fid*∗`)`
	|	`(infer)`

3. We could automate this procedure in the future. There is only one subtlety: it is hard to parse an empty list in Tree-sitter.

$$\begin{aligned}
&\mid\ \texttt{(dump)}\\
&\mid\ \texttt{(exit)}\\
\mathit{expr} &\rightarrow \texttt{(}\mathit{fid}\ \mathit{expr}{*}\texttt{)}\\
&\mid\ \texttt{(->use}\ \mathit{num}\texttt{)}\\
&\mid\ \mathit{num}\\
&\mid\ \mathit{vid}\\
\mathit{port} &\rightarrow \texttt{(}\mathit{pid}\ \mathit{num}\texttt{)}\\
\mathit{sid}, \mathit{fid}, \mathit{pid}, \mathit{vid} &\rightarrow \mathit{any\text{-}character\text{-}except\text{-}parenthesis\text{-}and\text{-}spaces}{+}
\end{aligned}$$

Hopefully the semantics is mostly self-explanatory. In `defscope`, the first list of ports after `:import` is what port 0 of *fid* imports, and so forth; each pair of ports after `:bind` means the first one can access the second one. `infer` triggers scope inference and, if succeeds, clears all pending macros. `regexp` outputs two regular expressions R and D, containing only function parameters listed in the argument. `dump` outputs everything in the same syntax, including sorts, functions, scope rules, and macros (if any). The syntax of regular expression is listed next.

$$\begin{aligned}
\mathit{regexp} &\rightarrow \texttt{(concat}\ \mathit{regexp}{*}\texttt{)}\\
&\mid\ \texttt{(or}\ \mathit{regexp}{*}\texttt{)}\\
&\mid\ \texttt{(many}\ \mathit{regexp}\texttt{)}\\
&\mid\ \mathit{fid}\,.\,\mathit{pid}[\uparrow\downarrow]
\end{aligned}$$

Regular expressions are obtained from NFAs using Brzozowski's algebraic method ([BM63]). They are largely unsimplified, so might not be legible enough to serve as documentation. We left the various applications in Section 4.2, and specifically the soundness check of `->use`, as future work.

Here we show time and memory usage of examples in this article. Data is collected on a machine with 8 × 11th Gen Intel® Core™ i5-11300H and 15.4 GiB RAM.

program	time (ms)	memory (peak, MB)	number of vars	number of clauses
`letrec`	0.37	4.34	185	336
`do`	0.46	4.35	229	241
`match`	2.09	4.86	2433	5763
`let`	0.22	4.34	54	55
`F`	0.11	0.09	/	/
`logic`	0.11	0.09	/	/

Table 2. Efficiency test result. The last two examples do not contain `(infer)`

6 Future and Related Work

Theories

Thanks to the regular-language characterization, we formalize some research topics in terms of formal languages.

1. The decision problem in Conjecture 12 appears to be hard. Is there any easily decidable criteria of R, maybe with the help of user-provided annotations, that ensure the existence of L_1? That would correspond to a sound type system for macros. [DKRW15] gives a closely related result: the classes of regular languages and Church-Rosser congruential languages are equivalent. However, the latter poses a overly strong requirement for the rewriting relation.
2. Are there better heuristic algorithms to compute L_1? The current SAT-solving one is straightforward: it simply enumerate the number of states of the NFA and decide if the number is enough.
3. Do other classes of languages, such as context-free languages or TreeReg ([Meh17]), have intuitive explanations in terms of scope rules, such as through push-down automata? Since NPDAs and DPDAs are not equivalent, it would be interesting to know which kinds of scopes are lost for determinacy. We can also take about paths instead of simple paths, which might have better composibility.
4. Can we revert regular expressions back to scope rules? Yes, we can always get a finite automaton, but what if the number of ports are given? Users may not write complex regular expressions by hand, but modifying existing ones sounds tractable.

We explored the first two directions posted in the last paragraph of [PKW17]. The third direction—supporting modules—is more challenging. There has been a lot of progress on scope graphs since 2017 ([ZvA23]), from where we can take inspirations. The combination of list patterns and variable number of ports seems more straightforward.

Implementation

Along the lines of Ott, we can derive helper functions from scope rules. In particular, we can generate `substitution` automatically for Osazone, which must be hardcoded together with sugar definitions.

Along the lines of Statix and NaBL, we can develop more efficient name-resolution algorithms, especially whole-program ones. Possible improvements include (1) optimizing away useless dynamic queries through static scope rules, (2) reusing previous resolution information by designing an incremental algorithm, and (3) utilizing parallelism. It would be best to integrate into Tree-sitter so that we can reuse its incremental parser and user interface.

Related work

[romeo, nominal logic, scope graph, ott, binding as sets of scopes, A Theory of Hygienic Macros]

7 Conclusion

Scopes and bindings are artificial. Humans, not the nature, postulate how scopes in a programming language works. Comparing to semantics of programming languages, scoping rules are like the "frontend", with an emphasis on HCI rather than some logical or mathematical nature. Most researches, frameworks, and theories on names and name resolution do not satisfy everyone. This situation, however, is quite similar to DSLs: while some DSLs (SQL, some PPLs, etc.) are intentionally restrictive to permit more efficient execution or inference, others—definable using sugars—are designed to give a cleaner, prettier frontend for end users. I am not saying researching into the two are unnecessary or meaningless; instead, I think it is important, but its significance may only be verified by practical use experiences.

This article is not an exception. With a small extension to the binding language, we get a exact and robust characterization of what we can model, without losing much "inferability". However, there are many things we cannot model, and whether this class of scopes are efficiently executable is still unclear. We possess a positive attitude, because of great properties and vast applicability of regular languages.

Acknowledgments

I appreciate Prof. Hu Zhenjiang and Prof. Cao Qinxiang for giving me the internship chance at Peking University. My life would have nothing to do with PKU otherwise.

I thank Guan Zhichao for building the OSAZONE programming system, and introducing it to me.

I thank Wang Ziheng for introducing me with inspirational papers on macros, scopes, and bindings.

I thank Tang Lijuan for introducing me with language interoperability.

I thank Yu Tailai for his insightful comments and ideas on various topics.

Bibliography

[BM63] J. A. Brzozowski and E. J. McCluskey. Signal flow graph techniques for sequential circuit state diagrams. *IEEE Transactions on Electronic Computers*, EC-12(2):67–76, 1963.

[DKRW15] Volker Diekert, Manfred Kufleitner, Klaus Reinhardt, and Tobias Walter. Regular languages are church-rosser congruential. *J. ACM*, 62(5), nov 2015.

[ES04] Niklas Eén and Niklas Sörensson. An extensible sat-solver. In Enrico Giunchiglia and Armando Tacchella, editors, *Theory and Applications of Satisfiability Testing*, pages 502–518. Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

[Fel91] Matthias Felleisen. On the expressive power of programming languages. *Sci. Comput. Program.*, 17(1–3):35–75, dec 1991.

[Fla16] Matthew Flatt. Binding as sets of scopes. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '16, pages 705–717. New York, NY, USA, 2016. Association for Computing Machinery.

[KD13] Andrew W. Keep and R. Kent Dybvig. A nanopass framework for commercial compiler development. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming*, ICFP '13, pages 343–350. New York, NY, USA, 2013. Association for Computing Machinery.

[Knu68] Donald E. Knuth. Semantics of context-free languages. *Mathematical systems theory*, 2(2):127–145, Jun 1968.

[Leu08] Peter Leupold. On regularity-preservation by string-rewriting systems. In Carlos Martín-Vide, Friedrich Otto, and Henning Fernau, editors, *Language and Automata Theory and Applications*, pages 345–356. Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[Mak77] G. S. Makanin. The problem of solvability of equations in a free semigroup. *Mathematics of The Ussr-sbornik*, 32:129–198, 1977.

[Meh17] Ben Mehne. Treeregex : an extension to regular expressions for matching and manipulating tree-structured text (technical report). 2017.

[MP96] M. Marcus and A. Pnueli. Using ghost variables to prove refinement. In Martin Wirsing and Maurice Nivat, editors, *Algebraic Methodology and Software Technology*, pages 226–240. Berlin, Heidelberg, 1996. Springer Berlin Heidelberg.

[NTVW15] Pierre Neron, Andrew Tolmach, Eelco Visser, and Guido Wachsmuth. A theory of name resolution. In Jan Vitek, editor, *Programming Languages and Systems*, pages 205–231. Berlin, Heidelberg, 2015. Springer Berlin Heidelberg.

[PK14] Justin Pombrio and Shriram Krishnamurthi. Resugaring: lifting evaluation sequences through syntactic sugar. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, pages 361–371. New York, NY, USA, 2014. Association for Computing Machinery.

[PK15] Justin Pombrio and Shriram Krishnamurthi. Hygienic resugaring of compositional desugaring. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming*, ICFP 2015, pages 75–87. New York, NY, USA, 2015. Association for Computing Machinery.

[PK18] Justin Pombrio and Shriram Krishnamurthi. Inferring type rules for syntactic sugar. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2018, pages 812–825. New York, NY, USA, 2018. Association for Computing Machinery.

[PKW17] Justin Pombrio, Shriram Krishnamurthi, and Mitchell Wand. Inferring scope through syntactic sugar. *Proc. ACM Program. Lang.*, 1(ICFP), aug 2017.

[SNO+10] PETER SEWELL, FRANCESCO ZAPPA NARDELLI, SCOTT OWENS, GILLES PESKINE, THOMAS RIDGE, SUSMIT SARKAR, and ROK STRNIŠA. Ott: effective tool support for the working semanticist. *Journal of Functional Programming*, 20(1):71–122, 2010.

[YXGH22] Ziyi Yang, Yushuo Xiao, Zhichao Guan, and Zhenjiang Hu. A lazy desugaring system for evaluating programs with sugars. In *Fuji International Symposium on Functional and Logic Programming*. 2022.

[ZvA23] Aron Zwaan and Hendrik van Antwerpen. Scope Graphs: The Story so Far. In Ralf Lämmel, Peter D. Mosses, and Friedrich Steimann, editors, *Eelco Visser Commemorative Symposium (EVCS 2023)*, volume 109 of *Open Access Series in Informatics (OASIcs)*, pages 32–1. Dagstuhl, Germany, 2023. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.