Romeo: a System For More Flexible Binding-Safe Programming

Paul Stansifer Mitchell Wand
Northeastern University
{pauls,wand}@ccs.neu.edu

Abstract

Current languages for safely manipulating values with names only support term languages with simple binding syntax. As a result, no tools exist to safely manipulate code written in those languages for which name problems are the most challenging. We address this problem with Romeo, a language that respects α-equivalence on its values, and which has access to a rich specification language for binding, inspired by attribute grammars. Our work has the complex-binding support of David Herman's λ_m, but is a full-fledged binding-safe language like Pure FreshML.

Categories and Subject Descriptors D.3.1 [*Formal Definitions and Theory*]

Keywords languages; binding; alpha-equivalence; macros

1. Introduction

Manipulating terms with binding information has traditionally posed a serious problem to metaprogrammers. In a survey of 9 DSL implementations, 8 were found to be prone to variable capture [4].

Building a system that does not suffer from this problem is difficult, because a name in isolation is meaningless; only its relationship with other names imbues a name with meaning. On the other hand, the meaninglessness of names inspires a motto for what it means to manipulate names correctly: a system is correct if α-conversion of input values results in outputs that are identical up to α-conversion [8].

Systems with this property have been created, allowing for programming with names but without insidious name problems. However, these systems generally support only term languages with simple binding structure [5, 12]. Our work is an extension of David Herman's macro system that supports complex binding structure [7], but as a full-fledged binding-safe language inspired by Pure FreshML [13].

1.1 Motivation

For example, consider the following Scheme term, exhibiting a complex binding structure:

```
(let* ((a 1)
       (b (+ a a))
       (c (* b 5)))
  (display c))
```

In Scheme, the `let*` syntactic form is defined to bind the names it introduces not only in the body, but also in the right hand side of each subsequent arm. Thus, this example has no free names, and the value of `c` is 10. Formally expressing these properties is the job of binding specifications, such as those in Nominal Isabelle [17], Ott [16], and Cαml [12].

If we want to programmatically manipulate source code while respecting its binding structure, we must add to our system a notion of α-equivalence. Only then can the correctness motto of preservation of α-equivalence be well-defined. We would like our notion of α-equivalence to be as compositional as possible, even though a pair of α-equivalent may decompose into pairs of subterms which are not α-equivalent. For example, our expression above is α-equivalent to the following:

```
(let* ((d 1)
       (d (+ d d))
       (d (* d 5)))
  (display d))
```

This is despite the fact that `(display d)` is clearly not α-equivalent to `(display c)`. The standard solution is to observe that they *are* α-equivalent after performing a substitution determined by examining the binders. This is our solution as well, but in order to model forms like `let*`, we must allow binders to be "exported" up from subforms (and sub-subforms, etc.) in a well-defined way. We say such binders are "buried." Furthermore, such a buried binder may be referred to inside the form that exports it, causing it to participate in multiple binding relationships at different levels.

It is instructive to contrast our work to another system capable of handling complex binding structure, the Dybvig algorithm [3] used in many Scheme implementations. In this algorithm, names are dynamically marked to indicate which macro evaluation they originated from, in order to prevent errors caused by coincidental name collision. We will discuss the limitations of this approach in section 7.4.

An improvement upon this is David Herman's λ_m-calculus [7], a macro system that uses binding specifications to guarantee that α-equivalent inputs will expand to α-equivalent outputs. However, its macros can be defined only in terms of pattern-matching. Pattern-matching systems are a natural way to define simple macros, but they lack the power to define more intricate macros, which are often the macros that benefit the most from access to complex binding constructs.

ICFP'14, September 1–6, 2014, Gothenburg, Sweden.

ACM 978-1-4503-2873-9/14/09…$15.00.
http://dx.doi.org/10.1145/2628136.2628162

CoreExpr ::=	RAtom	(*variable reference*)	var
|	Prod(*CoreExpr*, *CoreExpr*)	(*application*)	app
|	Prod(BAtom, *CoreExpr*↓0)	(*abstraction*)	lam
Expr ::=	RAtom	(*variable reference*)	var
|	Prod(*Expr*, *Expr*)	(*application*)	app
|	Prod(BAtom, *Expr*↓0)	(*abstraction*)	lam
|	Prod(*LetStarClauses*, *Expr*↓0)	(*sequential let*)	let-star
LetStarClauses ::=	Prod()	(*no clauses*)	lsc-none
|	$\text{Prod}^{\Uparrow(1 \triangleright 0)}(\text{Prod}^{\Uparrow 0}(\text{BAtom}, \textit{Expr}), \textit{LetStarClauses}{\downarrow}0)$	(*clause, and more clauses*)	lsc-none

Figure 1. Example types for two lambda calculi, one of which has the `let*` form. (The names from the right-hand column are used to identify injections and as the variables for cases in Figure 2.)

```
1  (define-fn (convert e:Expr) : CoreExpr
2    (case e
3      (var ⇒ (inj_var var))
4      (app ⇒ (open app (e1, e2) (inj_app (prod convert(e1), convert(e2)))))
5      (lam ⇒ (open lam (bv, e-body) (inj_lam (prod bv, convert(e-body)↓0))))
6      (let-star ⇒
7        (open let-star (lsc, e-body)
8          (case lsc
9            (lsc-none ⇒ convert(e-body))
10           (lsc-some ⇒
11             (open lsc-some (bv, val-expr, lsc-rest)
12               (let e-rest be convert((inj_let-star (prod lsc-rest, e-body↓0)))
13                 in (inj_app (prod (inj_lam (prod bv, e-rest↓0)), convert(val-expr))))))))))
```

Figure 2. A Romeo-L function to expand away `let*`

Our system is more flexible and can even operate outside the context of macro expansion altogether: we provide a full-fledged programming language for manipulating terms.

1.2 Example

As an example, we define in Figure 1 some types in our system, and in Figure 2 we define a function (using those types) that translates expressions from the lambda calculus augmented with a `let*` construct into the plain lambda calculus. We will discuss the meaning of the types in section 2.1, the behavior of the code in section 4.3, the way Romeo preserves α-equivalence for it in section 5.1, and the static guarantees that Romeo provides in section 6.3.

1.3 Contributions

Our primary contribution is an extension of David Herman's system for binding-safety in a pattern-matching macro system [7] to cover macros defined by procedures, and thus general meta-programming for terms with bindings. Our language is inspired by Pure FreshML [13].

Our system has the following features:

- Values in Romeo are "plain old data": atoms arranged in abstract syntax trees without binding information. Types provide the missing binding information.
- Romeo has an execution semantics which ensures that instead of a name "escaping" the context in which it is defined, a FAULT is produced.
- We prove a theorem guaranteeing that, in any execution, the dynamic environment can be replaced by one with α-equivalent values, and that execution will proceed to a value α-equivalent to what it otherwise would have.
- We provide a deduction system with which the programmer can establish that escape (and thus, FAULT) will never occur.

2. Binding language

2.1 Overview of binding types

Values in our system are plain old data, that is, S-expressions or something similar. We use binding types to specify the binding properties of these terms. Binding types augment a traditional context-free grammar with a single attribute (in the style of an attribute grammar) that represents the flow of bindings from one subterm to another.

In Figure 1, the type definition of *CoreExpr* looks like a traditional grammar for the lambda calculus, with one major difference: the notation *CoreExpr*↓0 indicates that the binder "exported" by the child in position 0 (the BAtom, which exports the name in that position) is to be in scope in the *CoreExpr* body of the lambda. To facilitate the connecting of binders to references, the type for names that bind, BAtom, is made distinct from the type of names that reference binders, RAtom.

Our system observes the convention that all names bound in a particular value are bound in all subvalues, unless overridden by a new binding for the same name. It is possible to imagine a system in which old names are removable (e.g., a construct `(unbind` x e`)`, in which the name x is not a valid reference in e, even if it was outside that construct), but this does not appear to be a feature that users are clamoring for. (But see the end-of-scope operator described by Hendriks and van Oostrom [6].)

A way to define the exports of a wide product is required when the binders are exported longer distances up the tree. Consider `let*`, in *Expr*. The "sequential let" line indicates that the binders exported by *LetStarClauses* are in scope in the body of the `let*` ex-

$$\begin{array}{rl} \textit{Expr} ::= \ldots & (\textit{same as before}) \\ \quad \mid \text{Prod}(\text{BAtom}, \text{BAtom}, \textit{Expr}{\downarrow}0 \uplus 1, \text{BAtom}, \textit{Expr}{\downarrow}0 \uplus 3) & (\textit{event handler}) \end{array}$$

Figure 3. Typed production rule for event handler

pression. The grammar for *LetStarClauses* says that a set of `let*`-clauses is either empty or a pair consisting of a single clause and the rest of the clauses. In the second production for *LetStarClauses*, the $\text{Prod}^{\Uparrow 0}(\text{BAtom}, \textit{Expr})$ indicates that the first clause exports the binder from its position 0 (that is, the BAtom). However, this name is not in scope in the *Expr*. The $\textit{LetStarClauses}{\downarrow}0$ indicates that the names exported from the first clause are in scope in the remainder of the clauses.

The $\Uparrow (1 \rhd 0)$ indicates that the entire set of clauses exports all the binders exported by either the first clause or the rest of the clauses, and that names in the rest of the clauses override those from the first clause. If we had wanted to specify that all the binders in a `let*` must be distinct, then we could have written $\Uparrow (1 \uplus 0)$, which behaves like $\Uparrow (1 \rhd 0)$, except that duplicated atoms are an error.

Thus we have an attribute grammar with a single attribute, whose values are sets of names representing bindings. These sets are synthesized from binders and inherited by other terms until they flow to references.

Our notations $\downarrow$, $\Uparrow$, $\rhd$, and $\uplus$ form an algebra of attributes; the tractability of this algebra is a key to many of our results. We call the terms in this language *binding combinators*.

2.1.1 Example: multiple, partially-shared bindings

For another example, imagine constructing a pair of event handlers, one of which handles mouse events and one of which handles keyboard events, but both of which need to know what GUI element is focused. This new form, defined in Figure 3, binds three atoms (the BAtoms, which are in positions 0, 1, and 3), one of which is bound in both subexpressions, and two of which are bound in only one of them. Here is a possible use of this new form:

```
(handler gui-elt
         mouse-evt (deal-with gui-elt mouse-evt)
         kbd-evt (tag gui-elt (text-of kbd-evt)))
```

And here is an α-equivalent, but harder-to-read, version:

```
(handler a
         b (deal-with a b)
         b (tag a (text-of b)))
```

The scope of the first `b` is the `(deal-with ...)`, and the scope of the second one is the `(tag ...)`.

Regardless of whether they have the same names, the meanings of the two events must not be conflated, but the GUI element must not be differentiated. For this reason, the operations our system performs on products must handle binding by first identifying what names are exported by each child (e.g. a BAtom or a Prod with a non-empty $\Uparrow$), and then determining which names are imported by which children. The latter is the responsibility of the $\downarrow$ operator.

Our goal of supporting realistic concrete syntax is particularly relevant here. The `handler` statement could be implemented as a function that gets called with a lambda (binding `gui-elt`) that returns a pair of lambdas (binding the `-evts`), at the cost of some inconvenience for the programmer. If the only binding construct in a language were lambda, an "off-the-shelf" nominal logic system would suffice as a basis for Romeo. However, programmer convenience is precisely the point of metaprogramming systems.

2.2 Binding types, in more detail

In this section, we introduce our actual language of binding types and the metalanguage we use to describe them.

$$\begin{array}{rl} a \in \textit{Atom} & \\ v \in \textit{Value} ::= & a \\ & \mid \mathbf{inj0}(v) \\ & \mid \mathbf{inj1}(v) \\ & \mid \mathbf{prod}_i(v_i) \end{array} \qquad \begin{array}{rl} \tau \in \textit{Type} ::= & \text{BAtom} \\ & \mid \text{RAtom} \\ & \mid \tau + \tau \\ & \mid \text{Prod}_i^{\Uparrow\beta}(\tau_i{\downarrow}\beta_i) \\ & \mid \mu X.\tau \\ & \mid X \end{array}$$

Values are either atoms, left- or right- injections of values (to model sum types), or tuples of values. We write $\mathbf{prod}_i(v_i)$ for the tuple $(v_0, \ldots, v_n)$, for some n. We will use notation like this for sequence comprehensions throughout our presentation.

The basic types are BAtom (for binders) and RAtom (for references). These types tell us how to interpret atoms. By convention, BAtoms export themselves and RAtoms export nothing.

Tuples are interpreted by Prod types. The wide product type $\text{Prod}^{\Uparrow\beta_{\text{ex}}}(\tau_0{\downarrow}\beta_0, \ldots, \tau_n{\downarrow}\beta_n)$, which we denote by the comprehension $\text{Prod}_i^{\Uparrow\beta_{\text{ex}}}(\tau_i{\downarrow}\beta_i)$, tells us how to interpret the value $\mathbf{prod}_i(v_i)$. The term β_i, constructed in our algebra of attributes, combines (some of) the binders exported by $v_0, \ldots, v_n$ to determine the local names bound in v_i. Again, by convention, these names override those inherited from outside ("above") $\mathbf{prod}_i(v_i)$. The binding combinator β_{ex}, similarly constructed in our algebra of attributes, combines the binders exported by $v_0, \ldots, v_n$ to determine the names exported as binders by the tuple $\mathbf{prod}_i(v_i)$.

To sidestep issues of parsing, we have sum types and injections. A value $\mathbf{inj0}(v)$ (resp. $\mathbf{inj1}(v)$) is interpreted by the type $\tau_0 + \tau_1$ so that v is interpreted by τ_0 (resp. τ_1).

Last, we have recursive types $\mu X.\tau$, where τ must be productive; to interpret a value v according to $\mu X.\tau$ is to interpret it according to $\tau\,[\mu X.\tau/X]$.

2.3 The algebra of binding combinators

Binding combinators are terms built from the following grammar:

$$\begin{array}{rl} i, \ell \in & \mathbb{N} \\ \beta \in \textit{Beta} ::= & \varnothing \\ & \mid \beta \uplus \beta \\ & \mid \beta \rhd \beta \\ & \mid \ell \end{array}$$

As discussed above, we use binding combinators to collect names from the sets exported by the subterms of a sequence $\mathbf{prod}_i(v_i)$. We will need to interpret these combinators over both sets of names and substitutions (finite maps from names to names). As before, we make liberal use of comprehensions: we write $[\![\beta]\!]\,(A_i)_i$ for $[\![\beta]\!]\,(A_0, \ldots, A_n)$, etc. The interpretation is as follows:

$$\begin{array}{c} [\![\_]\!]\,(\_) : \textit{Beta} \times \overline{\textit{AtomSet}} \to \textit{AtomSet} \\ [\![\varnothing]\!]\,(A_i)_i \triangleq \varnothing \\ [\![\ell]\!]\,(A_i)_i \triangleq A_\ell \\ [\![\beta \rhd \beta']\!]\,(A_i)_i \triangleq [\![\beta]\!]\,(A_i)_i \cup [\![\beta']\!]\,(A_i)_i \\ [\![\beta \uplus \beta']\!]\,(A_i)_i \triangleq [\![\beta]\!]\,(A_i)_i \uplus [\![\beta']\!]\,(A_i)_i \end{array}$$

Here and elsewhere, we write $\overline{X}$ to mean a sequence of Xs.

A substitution σ is a partial function from atoms to atoms. For the purposes of manipulating them, we represent substitutions as a set of ordered pairs of atoms. Our substitutions are naïve, which

is to say that they ignore binding structure and simply affect all names. We interpret β's on substitutions as follows:

$$\begin{aligned}
[\![_]\!]\,(_) &: \mathit{Beta} \times \overline{\mathit{Subst}} \to \mathit{Subst} \\
[\![\varnothing]\!]\,(\sigma_i)_i &\triangleq \varnothing \\
[\![\ell]\!]\,(\sigma_i)_i &\triangleq \sigma_\ell \\
[\![\beta \rhd \beta']\!]\,(\sigma_i)_i &\triangleq [\![\beta]\!]\,(\sigma_i)_i \rhd [\![\beta']\!]\,(\sigma_i)_i \\
[\![\beta \uplus \beta']\!]\,(\sigma_i)_i &\triangleq [\![\beta]\!]\,(\sigma_i)_i \uplus [\![\beta']\!]\,(\sigma_i)_i
\end{aligned}$$

In this definition, $\uplus$ is disjoint union, undefined if the sets are not disjoint, and $\sigma \rhd \sigma'$ is defined as follows:

$$\sigma \rhd \sigma' \triangleq \sigma \cup \left\{ \langle a_\mathrm{d}, a_\mathrm{r} \rangle \;\middle|\; \begin{array}{l} \langle a_\mathrm{d}, a_\mathrm{r} \rangle \in \sigma' \\ a_\mathrm{d} \notin \mathrm{dom}(\sigma) \end{array} \right\}$$

Using $[\![\beta]\!]\,(A_i)_i$, we can compute the binders exported from any value. We call these the *free binders* of the value. As suggested above, the free binders of a value are determined using the type.

$$\begin{aligned}
\mathrm{fb}\Big(\mathrm{Prod}_i^{\Uparrow\beta_\mathrm{exp}}(\tau_i{\downarrow}\beta_i), \mathbf{prod}_i(v_i)\Big) &::= [\![\beta_\mathrm{exp}]\!]\,(\mathrm{fb}(\tau_i, v_i))_i \\
\mathrm{fb}(\tau_0 + \tau_1, \mathbf{inj0}(v)) &::= \mathrm{fb}(\tau_0, v) \\
\mathrm{fb}(\tau_0 + \tau_1, \mathbf{inj1}(v)) &::= \mathrm{fb}(\tau_1, v) \\
\mathrm{fb}(\mu X.\tau, v) &::= \mathrm{fb}(\tau[\mu X.\tau/X], v) \\
\mathrm{fb}(\mathrm{BAtom}, a) &::= \{a\} \\
\mathrm{fb}(\mathrm{RAtom}, a) &::= \varnothing
\end{aligned}$$

There are several other useful quantities that we can compute using these combinators. First is the set of free references of a term:

$$\begin{aligned}
\mathrm{fr}\Big(\mathrm{Prod}_i^{\Uparrow\beta_\mathrm{exp}}(\tau_i{\downarrow}\beta_i), \mathbf{prod}_i(v_i)\Big) &::= \bigcup_i \left(\begin{array}{l}\mathrm{fr}(\tau_i, v_i) \\ \setminus [\![\beta_i]\!]\,(\mathrm{fb}(\tau_j, v_j))_j\end{array}\right) \\
\mathrm{fr}(\tau_0 + \tau_1, \mathbf{inj0}(v)) &::= \mathrm{fr}(\tau_0, v) \\
\mathrm{fr}(\tau_0 + \tau_1, \mathbf{inj1}(v)) &::= \mathrm{fr}(\tau_1, v) \\
\mathrm{fr}(\mu X.\tau, v) &::= \mathrm{fr}(\tau[\mu X.\tau/X], v) \\
\mathrm{fr}(\mathrm{BAtom}, a) &::= \varnothing \\
\mathrm{fr}(\mathrm{RAtom}, a) &::= \{a\}
\end{aligned}$$

Next is the set of free atoms of a term, which is just the union of the free binders and the free references:

$$\mathrm{fa}(\tau, v) ::= \mathrm{fr}(\tau, v) \cup \mathrm{fb}(\tau, v)$$

Last is the set of exposable atoms, which are those non-free names that will become free when the value in question is broken into subterms. These are the atoms which are on their "last chance" for renaming before they become free. This set, only defined on products, is equal to the union of the binders exported by each term in a sequence, less the terms that are exported to the outside:

$$\begin{aligned}
&\mathrm{xa}\Big(\mathrm{Prod}_i^{\Uparrow\beta_\mathrm{exp}}(\tau_i{\downarrow}\beta_i), \mathbf{prod}_i(v_i)\Big) \\
&\quad ::= \left(\bigcup_i \mathrm{fb}(\tau_i, v_i)\right) \setminus \mathrm{fb}\Big(\mathrm{Prod}_j^{\Uparrow\beta_\mathrm{exp}}(\tau_j{\downarrow}\beta_j), \mathbf{prod}_j(v_j)\Big)
\end{aligned}$$

It is also useful to know the support of a binding combinator β:

$$\begin{aligned}
_ \hat{\in} _ &\subseteq \mathbb{N} \times \mathit{Beta} \\
\ell \hat{\notin} \varnothing &\triangleq \text{false} \\
\ell \hat{\in} \ell' &\triangleq \ell = \ell' \\
\ell \hat{\in} \beta \rhd \beta' &\triangleq \ell \hat{\in} \beta \text{ or } \ell \hat{\in} \beta' \\
\ell \hat{\in} \beta \uplus \beta' &\triangleq \ell \hat{\in} \beta \text{ or } \ell \hat{\in} \beta'
\end{aligned}$$

3. Alpha-equivalence

Our next task is to go from a binding type to a notion of α-equivalence on values described by that type. Because our binding types allow for buried binders (i.e., binders that may be an arbitrary depth from the form that binds them) to be exported, we define two values to be α-equivalent if both

- they export identical bindings, and
- local (non-exported) bindings can be renamed along with the names that reference them to make the terms identical.

We use $=_\mathrm{B}$ (pronounced "binder-equivalent") for the first relation and $=_\mathrm{R}$ (pronounced "reference-equivalent") for the second.

$$\_ =_\alpha \_ : \_ \subseteq \mathit{Value} \times \mathit{Value} \times \mathit{Type}$$

$$\frac{v =_\mathrm{B} v' : \tau \qquad v =_\mathrm{R} v' : \tau}{v =_\alpha v' : \tau}\ \alpha\mathrm{EQ}$$

3.1 Binder equivalence

Two values are $=_\mathrm{B}$ iff their exported (free) binders in the same positions are identical (references are irrelevant). Note that Bα-PROD examines only the subterms that are in the support of β_ex, because non-exported binders are the responsibility of $=_\mathrm{R}$.

Here and throughout, we omit the rules for injections and fixed points, which are trivial.

$$\_ =_\mathrm{B} \_ : \_ \subseteq \mathit{Value} \times \mathit{Value} \times \mathit{Type}$$

$$\frac{}{a =_\mathrm{B} a : \mathrm{BAtom}}\ \mathrm{B}\alpha\text{-BATOM} \qquad \frac{}{a =_\mathrm{B} a' : \mathrm{RAtom}}\ \mathrm{B}\alpha\text{-RATOM}$$

$$\frac{\forall i \,\hat{\in}\, \beta_\mathrm{ex}.\ v_i =_\mathrm{B} v_i' : \tau_i}{\mathbf{prod}_i(v_i) =_\mathrm{B} \mathbf{prod}_i(v_i') : \mathrm{Prod}_i^{\Uparrow\beta_\mathrm{ex}}(\tau_i{\downarrow}\beta_i)}\ \mathrm{B}\alpha\text{-PROD}$$

3.2 Reference equivalence

Calculating $=_\mathrm{R}$ is analogous to the conventional notion of α-equivalence, except that we need to extract and rename the bindings that are buried in subterms.

3.2.1 Joining the binders

We begin with the $\bowtie$ operator (pronounced "join"). It walks through both values in lockstep, collecting pairs of corresponding binding atoms and assigning a common fresh atom for each. The result is a pair of injective substitutions whose domains are equal to the set of free binders of the values being joined.

At a product, we do the following on each side: we first walk through each subterm, recursively generating substitutions for each binding exported by any of the subterms, making sure that there is no overlap between the fresh names (i.e. the ranges of the substitutions) assigned in different subterms. The substitutions for the subterms are then combined by β_ex to produce a substitution for the exported binders of the product term. These two substitutions (the last two terms of the $\bowtie$ relation) are the output of this relation.

We define $\#$ to be the disjointness operator over names, sets of names, and values. It is naïve, meaning that it entirely ignores binding structure. Therefore, $a \# \lambda b.b$ is true, but $a \# \lambda a.a$ is false.

$$\_ \bowtie \_ : \_ \to \_ \bowtie \_ \subseteq \mathit{Value} \times \mathit{Value} \times \mathit{Type} \times \mathit{Subst} \times \mathit{Subst}$$

$$\frac{}{a \bowtie a' : \mathrm{BAtom} \to \{\langle a, a_\mathrm{fresh}\rangle\} \bowtie \left\{\left\langle a', a_\mathrm{fresh}\right\rangle\right\}}\ \text{J-BATOM}$$

$$\frac{}{a \bowtie a' : \mathrm{RAtom} \to \varnothing \bowtie \varnothing}\ \text{J-RATOM}$$

$$\frac{\begin{array}{cc}\forall i.\ v_i \bowtie v_i' : \tau \to \sigma_i \bowtie \sigma_i' & \forall i \neq j.\ \mathrm{rng}(\sigma_i) \# \mathrm{rng}(\sigma_j) \\ \sigma = [\![\beta_\mathrm{ex}]\!]\,(\sigma_i)_i & \sigma' = [\![\beta_\mathrm{ex}]\!]\,\left(\sigma_i'\right)_i\end{array}}{\mathbf{prod}_i(v_i) \bowtie \mathbf{prod}_i(v_i') : \mathrm{Prod}_i^{\Uparrow\beta_\mathrm{ex}}(\tau_i{\downarrow}\beta_i) \to \sigma \bowtie \sigma'}\ \text{J-PROD}$$

For example, consider the two `let*` expressions we have previously discussed:

```
(let* ((a 1)          (let* ((d 1)
       (b (+ a a))           (d (+ d d))
       (c (* b 5)))          (d (* d 5)))
  (display c))          (display d))
```

The results of $\bowtie$ on their children in position 1 (the `display` expressions) are $\varnothing$ and $\varnothing$, because neither one has any free binders. Position 0 corresponds to the *LetStarClauses*, and is more interesting. $\bowtie$ will nondeterministically generate three names, which we will choose to be aa, bb, and cc. Then, we will have $\sigma_0 = \{\langle \texttt{a}, \texttt{aa}\rangle, \langle \texttt{b}, \texttt{bb}\rangle, \langle \texttt{c}, \texttt{cc}\rangle\}$ and $\sigma_0' = \{\langle \texttt{d}, \texttt{cc}\rangle\}$. The different ranges of these substitutions indicate that some names (the ones called `a` and `b` in the left-hand value) cannot be referred to at all by references on the right-hand side, due to shadowing.

A more complete derivation of this is shown in Figure 4. The first step is straightforward: `c` and `d` are unified with each other, and the resulting substitutions are merged with the empty substitution. In the next step, `b` and `d` are unified, but the substitution in position 1 (which produced `cc`) takes precedence over the new definition of `d`. Finally, a similar process completes generating the substitutions corresponding to the free binders in each *LetStarClauses*.

3.2.2 Comparison by substitution

Now we can write the rules for $=_{\text{R}}$. At RAtom, the atoms being compared are necessarily free and must be identical in order to be reference-equal. Symmetrically to $=_{\text{B}}$, any two atoms are $=_{\text{R}}$ at BAtom.

At a product, the information from $\bowtie$ is used by $=_{\text{R}}$ to make the subterms comparable without requiring context. This is done as follows: For each pair of subterms v_i, v_i', we use $\bowtie$ to generate a pair of substitutions σ_i, σ_i' that rename the binders exported by v_i, v_i' to be identical. We then apply these renamings to the subterms, as directed by β_i.

Applying these substitutions to each pair of subterms (resulting in the new values $[\![\beta_i]\!]\,(\sigma_j)_j(v_i)$ and $[\![\beta_i]\!]\,(\sigma_j')_j(v_i')$) allows us to examine each pair of children in isolation. Note that this substitution is naïve (that is, it disregards types and therefore binding). Even though we are only interested in the substitution's effect on free references, this naïvete is acceptable because, first, $=_{\text{R}}$ does not examine free binders, and second, (broadly speaking) the substitution of un-free names is harmless (this principle is illustrated by Lemma 3.2).

Because our substitutions are naïve, we require that each substitution's range be disjoint from the values being examined. Without this requirement, we would have `(let* ((x 7)) x)` $=_{\text{R}}$ `(let* ((y 7)) a)` : *Expr*, witnessed by $\sigma_0 = \{\langle \texttt{x}, \texttt{a}\rangle\}$ and $\sigma_0' = \{\langle \texttt{y}, \texttt{a}\rangle\}$.

$$\_ =_{\text{R}} \_ : \_ \subseteq \mathit{Value} \times \mathit{Value} \times \mathit{Type}$$

$$\frac{}{a =_{\text{R}} a' : \text{BAtom}}\ \text{R}\alpha\text{-BAtom} \qquad \frac{}{a =_{\text{R}} a : \text{RAtom}}\ \text{R}\alpha\text{-RAtom}$$

$$\frac{\begin{array}{c}\forall i.\ v_i \bowtie v_i' : \tau_i \to \sigma_i \bowtie \sigma_i' \\ \forall i, j.\ \text{rng}(\sigma_i), \text{rng}(\sigma_i') \# v_j, v_j' \\ \forall i \neq j.\ \text{rng}(\sigma_i) \# \text{rng}(\sigma_j) \\ \forall i.\ [\![\beta_i]\!]\,(\sigma_j)_j(v_i) =_{\text{R}} [\![\beta_i]\!]\,(\sigma_j')_j(v_i') : \tau_i\end{array}}{\mathbf{prod}_i(v_i) =_{\text{R}} \mathbf{prod}_i(v_i') : \text{Prod}_{\tau\downarrow\beta}(\beta_{\text{ex}})}\ \text{R}\alpha\text{-Prod}$$

In our ongoing `let*` example, the appropriate substitution is a no-op on the *LetStarClauses*, which import nothing. Recursive application of $=_{\text{R}}$ will discover their shared binding structure and compare them as equal. On the other hand, the expression bodies will be both transformed into `(display cc)`, which lacks binding structure, and is naïvely equal to itself. So, the two expressions are $=_{\text{R}}$. (They are also trivially $=_{\text{B}}$, and therefore $=_\alpha$.) A complete derivation of their reference-equivalence, including analyzing the *LetStarClauses* themselves, is available at `http://hdl.handle.net/2047/d20005012`.

An example that better demonstrates the complexities of renaming is the event handler example from section 2.1.1. The result of invoking $\bowtie$ on each pair of children, in order to compare the two versions for α-equivalence, is in Figure 5.

All of the β's, except β_2 and β_4, are $\varnothing$. β_2 is $0 \uplus 1$ and β_4 is $0 \uplus 3$. The result of performing those substitutions is shown in Figure 6, establishing the relationship between `mouse-evt` and the first `b` and the relationship between `kbd-evt` and the second `b`.

3.3 Lemmas

The following lemmas establish that $=_\alpha$ has behavior consistent with an α-equivalence. Each follows from analogous lemmas about $=_{\text{B}}$ and $=_{\text{R}}$ (where fb and fr replace fa, if applicable).

Lemma 3.1. *$=_\alpha$ is an equivalence relation.*

Lemma 3.2 (Unfree atoms can be renamed)**.** *Suppose σ is injective and* $\text{rng}(\sigma) \# v$ *and* $\text{dom}(\sigma) \# \text{fa}(\tau, v), \text{rng}(\sigma)$. *Then* $\sigma(v) =_\alpha v : \tau$.

Lemma 3.3 (Good substitutions preserve α-equivalence)**.** *Suppose σ is injective and* $\text{rng}(\sigma) \# v, v'$ *and* $\text{dom}(\sigma) \# \text{rng}(\sigma)$. *Then* $v =_\alpha v' : \tau \Rightarrow \sigma(v) =_\alpha \sigma(v') : \tau$.

Lemma 3.4 (Free atoms are the same for α-equivalent values)**.** $v =_\alpha v' : \tau \Rightarrow \text{fa}(\tau, v) = \text{fa}(\tau, v')$

4. Romeo

Romeo is a first-order, typed, side-effect-free language whose values are abstract syntax trees. It uses types to direct the interpretation of these trees as syntax trees with binding, and to direct the execution of expressions in a way that respects that binding structure. There are three parts to this:

- First, the execution semantics ensures that whenever the program causes a name to escape the context in which it is defined, a FAULT is produced.
- Second, we provide theorems guaranteeing that at any point in execution, the dynamic environment could be replaced by one with α-equivalent values, and execution would still proceed to a value α-equivalent to what it otherwise would have. Furthermore, execution is deterministic up to α: that is, the nondeterministic choices that are made (e.g. for fresh identifiers) do not change the α-equivalence class of the result.
- Last, we provide a deduction system to generate proof obligations which, if satisfied, guarantee that escape (and thus, FAULT) will never occur.

The syntax of Romeo is given as follows:

$$\begin{array}{rcl}
p \in \mathit{Prog} & ::= & \mathit{fD} \ldots\ e : \tau \\
\mathit{fD} \in \mathit{FnDef} & ::= & (\textbf{define-fn}\ (f\ x : \tau \ldots\ \textbf{pre}\ C) : \tau\ e\ \textbf{post}\ C) \\
e \in \mathit{Expr} & ::= & (f\ x \ldots) \\
& | & (\textbf{fresh}\ x\ \textbf{in}\ e) \\
& | & (\textbf{let}\ x\ \textbf{where}\ C\ \textbf{be}\ e\ \textbf{in}\ e) \\
& | & (\textbf{case}\ x\ \ (x\ e)\ \ (x\ e)) \\
& | & (\textbf{open}\ x\ \ (x \ldots)\ e) \\
& | & (\textbf{if}\ x\ \textbf{equals}\ x\ e\ e) \\
& | & e^{\text{qlit}} \\
e^{\text{qlit}} \in \mathit{QuasiLit} & ::= & x \\
& | & (\textbf{ref}\ x) \\
& | & \left(\textbf{inj}_0\ e^{\text{qlit}}\ \tau\right) \\
& | & \left(\textbf{inj}_1\ \tau\ e^{\text{qlit}}\right) \\
& | & \left(\textbf{prod}_i^{\Uparrow\beta}\ e_i^{\text{qlit}}{\downarrow}\beta_i\right)
\end{array}$$

Here C ranges over a language of invariants from which the proof obligations for static safety are constructed. Romeo's operational semantics does not refer to these invariants. This sublanguage is discussed in Section 6.

Define: $v_{0,1,1} \triangleq$ `((c (* b 5)))` and $v'_{0,1,1} \triangleq$ `((d (* d 5)))`
$v_{0,1} \triangleq$ `((b (+ a a)) (c (* b 5)))` and $v'_{0,1} \triangleq$ `((d (+ d d)) (d (* d 5)))`
$v_0 \triangleq$ `((a 1) (b (+ a a)) (c (* b 5)))` and $v'_0 \triangleq$ `((d 1) (d (+ d d)) (d (* d 5)))`

(3.1)	$\texttt{c} \bowtie \texttt{d} : \text{BAtom} \to \{\langle \texttt{c}, \texttt{cc}\rangle\} \bowtie \{\langle \texttt{d}, \texttt{cc}\rangle\}$	J-BATOM
(3.2)	$\texttt{(* b 5)} \bowtie \texttt{(* d 5)} : \mathit{Expr} \to \varnothing \bowtie \varnothing$	no free bindings
(3.3)	$\texttt{(c (* b 5))} \bowtie \texttt{(d (* d 5))} : \text{Prod}^{\Uparrow 0}(\text{BAtom}, \mathit{Expr}) \to \{\langle \texttt{c}, \texttt{cc}\rangle\} \bowtie \{\langle \texttt{d}, \texttt{cc}\rangle\}$	J-PROD, 3.1 and 3.2
(3.4)	$v_{0,1,1} \bowtie v'_{0,1,1} : \mathit{LetStarClauses} \to \{\langle \texttt{c}, \texttt{cc}\rangle\} \bowtie \{\langle \texttt{d}, \texttt{cc}\rangle\}$	J-PROD, 3.3
(3.5)	$\texttt{b} \bowtie \texttt{d} : \text{BAtom} \to \{\langle \texttt{b}, \texttt{bb}\rangle\} \bowtie \{\langle \texttt{d}, \texttt{bb}\rangle\}$	J-BATOM
(3.6)	$\texttt{(+ a a)} \bowtie \texttt{(+ d d)} : \mathit{Expr} \to \varnothing \bowtie \varnothing$	no free bindings
(3.7)	$\texttt{(b (+ a a))} \bowtie \texttt{(d (+ d d))} : \text{Prod}^{\Uparrow 0}(\text{BAtom}, \mathit{Expr}) \to \{\langle \texttt{b}, \texttt{bb}\rangle\} \bowtie \{\langle \texttt{d}, \texttt{bb}\rangle\}$	J-PROD, 3.5 and 3.6
(3.8)	$[\![1 \triangleright 0]\!]\,(\{\langle \texttt{d}, \texttt{cc}\rangle\}, \{\langle \texttt{d}, \texttt{bb}\rangle\}) = \{\langle \texttt{d}, \texttt{cc}\rangle\}$	def. of $[\![]\!]$
(3.9)	$v_{0,1} \bowtie v'_{0,1} : \mathit{LetStarClauses} \to \{\langle \texttt{b}, \texttt{bb}\rangle, \langle \texttt{c}, \texttt{cc}\rangle\} \bowtie \{\langle \texttt{d}, \texttt{cc}\rangle\}$	J-PROD, 3.4, 3.7, and 3.8
(3.10)	$\texttt{a} \bowtie \texttt{d} : \text{BAtom} \to \{\langle \texttt{a}, \texttt{aa}\rangle\} \bowtie \{\langle \texttt{d}, \texttt{aa}\rangle\}$	J-BATOM
(3.11)	$\texttt{1} \bowtie \texttt{1} : \mathit{Expr} \to \varnothing \bowtie \varnothing$	no free bindings
(3.12)	$\texttt{(a 1)} \bowtie \texttt{(d 1)} : \text{Prod}^{\Uparrow 0}(\text{BAtom}, \mathit{Expr}) \to \{\langle \texttt{a}, \texttt{aa}\rangle\} \bowtie \{\langle \texttt{d}, \texttt{aa}\rangle\}$	J-PROD, 3.10 and 3.11
(3.13)	$[\![1 \triangleright 0]\!]\,(\{\langle \texttt{d}, \texttt{cc}\rangle\}, \{\langle \texttt{d}, \texttt{aa}\rangle\}) = \{\langle \texttt{d}, \texttt{cc}\rangle\}$	by def. of $[\![]\!]$
(3.14)	$v_0 \bowtie v'_0 : \mathit{LetStarClauses} \to \{\langle \texttt{a}, \texttt{aa}\rangle, \langle \texttt{b}, \texttt{bb}\rangle, \langle \texttt{c}, \texttt{cc}\rangle\} \bowtie \{\langle \texttt{d}, \texttt{cc}\rangle\}$	J-PROD, 3.12, 3.9, and 3.13

Figure 4. Example derivation of $\bowtie$ for *LetStarClauses*

j	v_j $\quad$ v'_j			σ_j $\quad$ σ'_j
0	`gui-elt` $\bowtie$ `a`	: BAtom	$\to$	$\{\langle \texttt{gui-elt}, \texttt{gg}\rangle\} \bowtie \{\langle \texttt{a}, \texttt{gg}\rangle\}$
1	`mouse-evt` $\bowtie$ `b`	: BAtom	$\to$	$\{\langle \texttt{mouse-evt}, \texttt{mm}\rangle\} \bowtie \{\langle \texttt{b}, \texttt{mm}\rangle\}$
2	`(deal-with gui-elt mouse-evt)` $\bowtie$ `(deal-with a b)`	: *Expr*	$\to$	$\varnothing \bowtie \varnothing$
3	`kbd-evt` $\bowtie$ `b`	: BAtom	$\to$	$\{\langle \texttt{kbd-evt}, \texttt{kk}\rangle\} \bowtie \{\langle \texttt{b}, \texttt{kk}\rangle\}$
4	`(tag gui-elt (text-of kbd-evt))` $\bowtie$ `(tag a (text-of b))`	: *Expr*	$\to$	$\varnothing \bowtie \varnothing$

Figure 5. Substitutions generated for the handler example

$[\![0 \uplus 1]\!]\,(\sigma_j)_j$`((deal-with gui-elt mouse-evt))` = `(deal-with gg mm)`
$[\![0 \uplus 1]\!]\,(\sigma'_j)_j$`((deal-with a b))` = `(deal-with gg mm)`
$[\![0 \uplus 3]\!]\,(\sigma_j)_j$`((apply-tag gui-elt (text-of kbd-evt)))` = `(apply-tag gg (text-of kk))`
$[\![0 \uplus 3]\!]\,(\sigma'_j)_j$`((apply-tag a (text-of b)))` = `(apply-tag gg (text-of kk))`

Figure 6. Result of substitution in the handler example

Typechecking is largely straightforward. In the body of **open**, the variables $x \ldots$ are given the types of the subterms of the scrutinee x, and in the body of **fresh**, x is bound to a new name at the type BAtom. In order to use that name as a reference, the **ref** form takes an argument of type BAtom and returns it as a RAtom.

We annotate injections with the types of the arm-not-taken, and product constructors with their binding structure. This allows us to write a function typeof (Γ, e) whose definition is routine.

4.1 Operational Semantics

We define Romeo's execution in big-step style.

We begin with auxiliary definitions that we will need:

$$\begin{array}{rcl} w \in \mathit{Result} & ::= & v \mid \text{FAULT} \\ \rho \in \mathit{ValEnv} & ::= & \epsilon \\ & \mid & \rho\,[z \to v] \\ \\ \Gamma \in \mathit{TypeEnv} & ::= & \epsilon \\ & \mid & \Gamma, z{:}\tau \end{array}$$

$$\text{fa}_{\text{env}}(\Gamma, \rho) = \bigcup_{x \in \text{dom}(\Gamma)} \text{fa}(\Gamma(x), \rho(x))$$

The form of the execution judgment is:

$$\Gamma \vdash_{\text{exe}} \langle e, \rho\rangle \overset{k}{\Longrightarrow} w$$

The k argument indicates the number of execution steps taken to produce the result in question.

Observe that some execution rules depend on the type environment Γ. This is because the binding structures of values are represented in their types (τ), but not in their runtime representations (v). Therefore, type erasure is not possible — the meaning of values (and thus the behavior of those rules) depends on type information.

We can now give the rules for execution in Romeo. Rules that introduce names come in two forms, -OK, and -ESCAPE. In each case, the only difference is that FAULT occurs in the -ESCAPE case. A fault indicates that a name has escaped the scope that created it (E-FRESH-$\star$) or exposed it (E-OPEN-$\star$). Much of the rest of the machinery in those rules is about ensuring that newly introduced names do not collide with each other or with names in the environment.

4.2 Execution rules

We begin with the rules for evaluating **fresh** expressions. The rules require that the new name not occur in the environment ρ. Our determinacy theorems (Theorems 5.1 and 5.2) guarantee that the choice of the new name will not affect the result (up to α-equivalence). We have two versions of the rule: FRESH-ESCAPE which returns FAULT when the new name appears in the result of executing the body e, and FRESH-OK, which returns w when that is not the case.

During the execution of e, x is treated as a BAtom. It is convertable to a RAtom by the expression (**ref** x).

The hypothesis $\tau = \text{typeof}((\Gamma, x{:}\text{BAtom}), e)$ is needed to synthesize the type τ in order to determine the free atoms of the result, in order to determine whether to produce FAULT or not. Determining the type, of course, is entirely static and could be precomputed once rather than at each evaluation.

$$\frac{\begin{array}{c}\tau = \text{typeof}((\Gamma, x{:}\text{BAtom}), e) \\ a \notin \text{fa}_{\text{env}}(\Gamma, \rho) \\ \Gamma, x{:}\text{BAtom} \vdash_{\text{exe}} \langle e, \rho[x \to a]\rangle \overset{k}{\Longrightarrow} w \\ w = \text{FAULT} \vee a \notin \text{fa}(\tau, w)\end{array}}{\Gamma \vdash_{\text{exe}} \langle (\textbf{fresh}\ x\ \textbf{in}\ e), \rho\rangle \overset{k+1}{\Longrightarrow} w}\ \text{E-FRESH-OK}$$

$$\frac{\begin{array}{c}\tau = \text{typeof}((\Gamma, x{:}\text{BAtom}), e) \\ a \notin \text{fa}_{\text{env}}(\Gamma, \rho) \\ \Gamma, x{:}\text{BAtom} \vdash_{\text{exe}} \langle e, \rho[x \to a]\rangle \overset{k}{\Longrightarrow} w \\ w \neq \text{FAULT} \wedge a \in \text{fa}(\tau, w)\end{array}}{\Gamma \vdash_{\text{exe}} \langle (\textbf{fresh}\ x\ \textbf{in}\ e), \rho\rangle \overset{k+1}{\Longrightarrow} \text{FAULT}}\ \text{E-FRESH-ESCAPE}$$

The next pair of rules deals with destructuring a product. Given a value $\rho(x) = \textbf{prod}(v_{\text{obj},0}, \ldots, v_{\text{obj},n})$, the **open** expression chooses an α-variant $\textbf{prod}(v_0, \ldots, v_n)$ and binds the resulting pieces to the variables x_i. In order to determine α-equivalence, the type τ is needed. In this way, types control the run-time behavior of Romeo programs. The names in the α-variant must be distinct both from names in the environment ρ and from each other (to the extent that they are not actually related by binding). This is taken care of by a subsidiary judgment $\vdash_{\text{suff-disj}}$. See section 4.2.2 for a more detailed discussion.

As with **fresh**, we have two rules which branch on whether any of the new names appear in the result of the body e. We test for escaped names by comparing the free atoms in the result with the exportable atoms of the renamed input. This suffices for safety because the only atoms that can become free are those that are exportable.

$$\frac{\begin{array}{c}\rho(x_{\text{obj}}) =_\alpha \textbf{prod}_i(v_i) : \tau_{\text{obj}} \qquad \Gamma \vdash_{\text{type}} x_{\text{obj}} : \tau_{\text{obj}} \\ \tau_{\text{obj}} = \text{Prod}_i^{\Uparrow \beta_{\text{exp}}}(\tau_i{\downarrow}\beta_i) \qquad \tau = \text{typeof}(\Gamma, e) \\ \text{fa}_{\text{env}}(\Gamma, \rho) \vdash_{\text{suff-disj}} \textbf{prod}_i(v_i){:}\tau_{\text{obj}} \\ \Gamma, (x_i{:}\tau_i)_i \vdash_{\text{exe}} \langle e, \rho[x_i \to v_i]_i\rangle \overset{k}{\Longrightarrow} w \\ w = \text{FAULT} \vee \text{xa}(\tau_{\text{obj}}, \textbf{prod}_i(v_i)) \# \text{fa}(\tau, w)\end{array}}{\Gamma \vdash_{\text{exe}} \langle (\textbf{open}\ x_{\text{obj}}\ ((x_i)_i)\ e), \rho\rangle \overset{k+1}{\Longrightarrow} w}\ \text{E-OPEN-OK}$$

$$\frac{\begin{array}{c}\rho(x_{\text{obj}}) =_\alpha \textbf{prod}_i(v_i) : \tau_{\text{obj}} \qquad \Gamma \vdash_{\text{type}} x_{\text{obj}} : \tau_{\text{obj}} \\ \tau_{\text{obj}} = \text{Prod}_i^{\Uparrow \beta_{\text{exp}}}(\tau_i{\downarrow}\beta_i) \qquad \tau = \text{typeof}(\Gamma, e) \\ \text{fa}_{\text{env}}(\Gamma, \rho) \vdash_{\text{suff-disj}} \textbf{prod}_i(v_i){:}\tau_{\text{obj}} \\ \Gamma, (x_i{:}\tau_i)_i \vdash_{\text{exe}} \langle e, \rho[x_i \to v_i]_i\rangle \overset{k}{\Longrightarrow} w \\ w \neq \text{FAULT} \wedge \neg(\text{xa}(\tau_{\text{obj}}, \textbf{prod}_i(v_i)) \# \text{fa}(\tau, w))\end{array}}{\Gamma \vdash_{\text{exe}} \langle (\textbf{open}\ x_{\text{obj}}\ ((x_i)_i)\ e), \rho\rangle \overset{k+1}{\Longrightarrow} \text{FAULT}}\ \text{E-OPEN-ESCAPE}$$

To simplify the deduction system, we require variables in some places where expressions would be more natural (like function arguments or x_{obj} in **open**). As a result, programs are written in (roughly) A-normal form, naming intermediate results with **let**. There are two evaluation rules for **let**, depending on whether calculating e_{val} faults. As noted avove, the constraint C is ignored at run-time.

$$\frac{\begin{array}{c}\Gamma \vdash_{\text{exe}} \langle e_{\text{val}}, \rho\rangle \overset{k_{\text{val}}}{\Longrightarrow} v_{\text{val}} \qquad \tau_{\text{val}} = \text{typeof}(\Gamma, e_{\text{val}}) \\ \Gamma, x{:}\tau_{\text{val}} \vdash_{\text{exe}} \langle e_{\text{body}}, \rho[x \to v_{\text{val}}]\rangle \overset{k_{\text{body}}}{\Longrightarrow} w\end{array}}{\Gamma \vdash_{\text{exe}} \langle (\textbf{let}\ x\ \textbf{where}\ C\ \textbf{be}\ e_{\text{val}}\ \textbf{in}\ e_{\text{body}}), \rho\rangle \overset{k_{\text{val}}+k_{\text{body}}+1}{\Longrightarrow} w}\ \text{E-LET}$$

$$\frac{\Gamma \vdash_{\text{exe}} \langle e_{\text{val}}, \rho\rangle \overset{k}{\Longrightarrow} \text{FAULT}}{\Gamma \vdash_{\text{exe}} \langle (\textbf{let}\ x\ \textbf{where}\ C\ \textbf{be}\ e_{\text{val}}\ \textbf{in}\ e_{\text{body}}), \rho\rangle \overset{k+1}{\Longrightarrow} \text{FAULT}}\ \text{E-LET-FAIL}$$

As in Pure FreshML [13], we assume that our expressions are evaluated in a context of function definitions, so that from a function name we can retrieve the function's formals and body. Since this context is constant throughout an execution, it is elided in the evaluation judgment.

$$\frac{\begin{array}{c}\text{body}(f) = e \qquad \text{formals}(f) = (x_{\text{formal},i}{:}\tau_{\text{formal},i})_i \\ (x_{\text{formal},i}{:}\tau_{\text{formal},i})_i \vdash_{\text{exe}} \langle e, [[x_{\text{formal},i} \to \rho_i\,(x_{\text{actual},i})]_i]\rangle \overset{k}{\Longrightarrow} w\end{array}}{\Gamma \vdash_{\text{exe}} \langle (f\ (x_{\text{actual},i})_i), \rho\rangle \overset{k+1}{\Longrightarrow} w}\ \text{E-CALL}$$

The remainder of the rules are routine. For simplicity's sake, the equality test construct works only on atoms.

$$\frac{\begin{array}{c}\rho(x_{\text{obj}}) = \textbf{inj0}(v_0) \qquad \Gamma(x_{\text{obj}}) = \tau_0 + \tau_1 \\ \Gamma, x_0{:}\tau_0 \vdash_{\text{exe}} \langle e_0, \rho[x_0 \to v_0]\rangle \overset{k}{\Longrightarrow} w\end{array}}{\Gamma \vdash_{\text{exe}} \langle (\textbf{case}\ x_{\text{obj}}\ (x_0\ e_0)\ (x_1\ e_1)), \rho\rangle \overset{k+1}{\Longrightarrow} w}\ \text{E-CASE-LEFT}$$

$$\frac{\begin{array}{c}\rho(x_{\text{obj}}) = \textbf{inj1}(v_1) \qquad \Gamma(x_{\text{obj}}) = \tau_0 + \tau_1 \\ \Gamma, x_1{:}\tau_1 \vdash_{\text{exe}} \langle e_1, \rho[x_1 \to v_1]\rangle \overset{k}{\Longrightarrow} w\end{array}}{\Gamma \vdash_{\text{exe}} \langle (\textbf{case}\ x\ (x_0\ e_0)\ (x_1\ e_1)), \rho\rangle \overset{k+1}{\Longrightarrow} w}\ \text{E-CASE-RIGHT}$$

$$\frac{\begin{array}{c}\rho(x_1) = a \qquad \rho(x_r) = b \qquad a = b \\ \Gamma \vdash_{\text{exe}} \langle e_0, \rho\rangle \overset{k}{\Longrightarrow} w\end{array}}{\Gamma \vdash_{\text{exe}} \langle (\textbf{if}\ x_l\ \textbf{equals}\ x_r\ e_0\ e_1), \rho\rangle \overset{k+1}{\Longrightarrow} w}\ \text{E-IF-YES}$$

$$\frac{\begin{array}{c}\rho(x_l) = a \qquad \rho(x_r) = b \qquad a \neq b \\ \Gamma \vdash_{\text{exe}} \langle e_1, \rho\rangle \overset{k}{\Longrightarrow} w\end{array}}{\Gamma \vdash_{\text{exe}} \langle (\textbf{if}\ x_l\ \textbf{equals}\ x_r\ e_0\ e_1), \rho\rangle \overset{k+1}{\Longrightarrow} w}\ \text{E-IF-NO}$$

The E-PROG rule initiates evaluation of a program. It is notionally responsible for setting up the (non-notated) function context.

$$\frac{\epsilon \vdash_{\text{exe}} \langle e, \epsilon\rangle \overset{k}{\Longrightarrow} v}{\vdash_{\text{exe}} fD \ldots\ e \overset{k+1}{\Longrightarrow} v}\ \text{E-PROG}$$

4.2.1 Quasi-Literals

The last language component is a category called *quasi-literals*, so called because they look like literal syntax for object-level syntax objects, except that they contain variable references (which denote values), not literal atoms. Of course, those variables may refer to atom values generated by **fresh**. Quasi-literals also contain some type information to avoid the need for type inference.

$$\frac{v = [\![e^{\text{qlit}}]\!]_\rho}{\Gamma \vdash_{\text{exe}} \langle e^{\text{qlit}}, \rho\rangle \overset{1}{\Longrightarrow} v}\ \text{E-QLIT}$$

Their evaluation is specified by the following rules:

$$\begin{aligned}
[\![_]\!]_{_} &: \mathit{QuasiLit} \times \mathit{ValEnv} \to \mathit{Value} \\
[\![x]\!]_\rho &= \rho(x) \\
[\![(\mathbf{ref}\ x)]\!]_\rho &= \rho(x) \\
[\![(\mathbf{inj}_0\ e^{\text{qlit}}\ \tau)]\!]_\rho &= \mathbf{inj0}([\![e^{\text{qlit}}]\!]_\rho) \\
[\![(\mathbf{inj}_1\ \tau\ e^{\text{qlit}})]\!]_\rho &= \mathbf{inj1}([\![e^{\text{qlit}}]\!]_\rho) \\
[\![\left(\mathbf{prod}_i^{\Uparrow\beta_{\text{exp}}}\ e_i^{\text{qlit}}{\downarrow}\beta_i\right)]\!]_\rho &= \mathbf{prod}_i([\![e_i^{\text{qlit}}]\!]_\rho)
\end{aligned}$$

4.2.2 Sufficient Disjointness

The requirement that evaluation be insensitive to α-equivalent inputs leads to strong requirements on the way that **open** destructures values. Consider the `let*` example above:

```
(let* ((a 1)          (let* ((d 1)
       (b (+ a a))           (d (+ d d))
       (c (* b 5)))          (d (* d 5)))
  (display c))          (display d)
```

These are α-equivalent, but if they were destructured without renaming, we would have $d = d = d$, even though $a \neq b \neq c$, violating our goal of being indifferent to α-conversion[1]. Therefore, we need to freshen each d to a distinct new name, e.g.

```
((aa 1)
 (bb (+ aa aa))
 (cc (* bb 5)))
```

The rule to ensure this is that, before destructuring, we must α-convert values such that the resulting renamed binders are disjoint from each other and from any names that appear in the environment. This gives rise to the hypothesis

$$\text{fa}_{\text{env}}(\Gamma, \rho) \vdash_{\text{suff-disj}} \mathbf{prod}_i(v_i){:}\tau_{\text{obj}}$$

in the E-OPEN-$\star$ rules. The $\vdash_{\text{suff-disj}}$ judgment here checks that binders exported by $\mathbf{prod}_i(v_i)$'s non-exported subterms are distinct from each other, and that the exposable atoms are disjoint from the free atoms in the environment.

To check the first part of that, we define the judgment

$$\vdash_{\text{bndrs-disj}} v : \tau,$$

which checks that the exported binders in v (as determined by the type τ) are disjoint from each other.

$$\frac{}{\vdash_{\text{bndrs-disj}} a : \text{BAtom}}\ \text{BD-BATOM}$$

$$\frac{}{\vdash_{\text{bndrs-disj}} a : \text{RAtom}}\ \text{BD-RATOM}$$

$$\frac{\begin{array}{c}\forall i, j \hat{\in} \beta_{\text{ex}}.\ i \neq j \Rightarrow \text{fb}(\tau_i, v_i)\ \#\ \text{fb}(\tau_j, v_j) \\ \forall i \hat{\in} \beta_{\text{ex}}.\ \vdash_{\text{bndrs-disj}} v_i : \tau_i\end{array}}{\vdash_{\text{bndrs-disj}} \mathbf{prod}_i(v_i) : \text{Prod}_i^{\Uparrow\beta_{\text{ex}}}(\tau_i{\downarrow}\beta_i)}\ \text{BD-PROD}$$

We can now define $\vdash_{\text{suff-disj}}$:

$$\frac{\begin{array}{c}\forall i \notin \beta_{\text{ex}}.\ \vdash_{\text{bndrs-disj}} v_i : \tau_i \\ \forall i, j \notin \beta_{\text{ex}}.\ i \neq j \Rightarrow \text{fb}(\tau_i, v_i)\ \#\ \text{fb}(\tau_j, v_j) \\ \text{xa}\left(\mathbf{prod}_i(v_i), \text{Prod}_i^{\Uparrow\beta_{\text{ex}}}(\tau_i{\downarrow}\beta_i)\right)\ \#\ A\end{array}}{A \vdash_{\text{suff-disj}} \mathbf{prod}_i(v_i){:}\text{Prod}_i^{\Uparrow\beta_{\text{ex}}}(\tau_i{\downarrow}\beta_i)}\ \text{SUFF-DISJ}$$

[1] In order to compare them, of course, the *LetStarClauses* would need to be destructured further. For simplicity, we consider the "last chance" for renaming to be the outermost level to which a name is exported, even if it is shadowed at that point.

4.3 Example

For an example, we write code that translates between two languages: from the lambda calculus augmented with a `let*` construct into the plain lambda calculus.

Our code, in Figure 2, mentions types defined in Figure 1. It is written in Romeo-L [10], which is a friendlier front-end to Romeo. For our purposes, the important differences are that the arguments to function calls and the scrutinees of **open** and **case** may be arbitrary expressions (not just variable references), and that Romeo-L can infer the strongest possible constraint C for **let**, so we may omit it. Furthermore, it will turn out (see section 6) that we need no pre- or post-condition from *convert* to show the absence of FAULT, so those constraints are also omitted.

Additionally, we have chosen to use more readable n-way sum types. This means that our **case** construct can branch 4 ways depending on whether the *Expr* it examines is a variable reference, an application, a lambda abstraction, or a let-star statement, and that injections take (as a subscript) a description of the choice that they are constructing.

Lines 3–5 are straightforward traversal of the existing *Expr* forms that are already forms in the core language (but, since their subterms might not be, they still need to be converted by recursively invoking *convert*).

Lines 6–11 destructure `let*` forms and handle the trivial case, where the `let*` does not have any arms. Line 12 recursively converts the body and all but the first arm of the `let*`, calling the result *e-rest*. Finally, line 13 constructs a beta-redex in the object language to bind the first arm's name to its value expression in *e-rest*.

5. Romeo respects α-equivalence

We are now ready to prove our main theorem: that Romeo respects α-equivalence. This is stronger than the corresponding result for Pure Fresh ML [13], which claims only that fresh names do not escape (see section 7.2 for a discussion).

Romeo is nondeterministic in its choice of names in the E-FRESH-$\star$ and E-OPEN-$\star$ rules. This complicates the definition of respecting α-equivalence. We show two results: first, that if two α-equivalent environments both terminate, then their results are α-equivalent, and second, if one of two α-equivalent environments yields a result, then the other one must yield at least one α-equivalent result as well.

For each of these theorems, the vast majority of the complexity is contained in the cases for E-FRESH-$\star$ and E-OPEN-$\star$ cases.

Since we must account for faulting, we extend the definition of α-equivalence to assert that FAULT $=_\alpha$ FAULT.

Complete proofs of all the theorems and lemmas we mention in this paper are available at `http://hdl.handle.net/2047/d20005013`.

Theorem 5.1 (Determinism up to α-equivalence, termination-insensitive version)**.**

$$\begin{array}{ll}\text{If} & \tau = \text{typeof}\,(\Gamma, e) \\ & \text{and } \rho =_\alpha \rho' : \Gamma \\ & \text{and } \Gamma \vdash_{\text{exe}} \langle e, \rho\rangle \overset{k}{\Longrightarrow} w \\ & \text{and } \Gamma \vdash_{\text{exe}} \langle e, \rho'\rangle \overset{k'}{\Longrightarrow} w' \\ \text{then} & w =_\alpha w' : \tau\end{array}$$

Proof. (Sketch) The major problem in the proof is that the two executions will potentially generate different fresh names in E-FRESH-$\star$ and E-OPEN-$\star$. Hence, even if the environments start out α-equivalent, they will not stay α-equivalent. For example, a **fresh** statement nondeterministically introduces a new free name into the environment. Therefore we must generalize our induction

hypothesis to account for the ways in which ρ and ρ' diverge from α-equivalence.

We account for this divergence by introducing two injective substitutions to unify the names introduce by the two executions. So our induction hypothesis says that if $\sigma \circ \rho =_\alpha \sigma' \circ \rho' : \Gamma$ for a pair of injective substitutions σ and σ', then the results will be α-equivalent, modulo the same transformation (i.e. $\sigma(w) =_\alpha \sigma'(w') : \tau$).

Consider the case of E-FRESH-$\star$. As we enter the scope of the new name it generates, σ and σ' are extended to map the new names to a common fresh name, for the sake of the induction hypothesis.

When we exit from the scope, the induction hypothesis tells us that the results (of the inductive evaluation) are equivalent modulo the *extended* substitutions. The results of this evaluation step are either the same as those of the inductive step, or FAULT. We first show that the *original* substitutions suffice to α-equate those two values, and then that one side faults if and only if the other side does.

The E-OPEN-$\star$ case proceeds with a similar structure. However, in this case, we are not generating a single pair of new names, but unpacking a pair of values, which potentially contain many names. The crucial lemma to handle this states that, given two α-equivalent (and $\vdash_{\text{suff-disj}}$) values, breaking them apart into their children results in pairs of values which are pairwise α-equivalent modulo a *single* pair of substitutions. In other words, the subterms of a sufficiently disjoint value can all be placed into the same environment without losing any binding information (which would happen if there were any name collisions). After the induction hypothesis, E-OPEN-$\star$ proceeds like E-FRESH-$\star$.

The E-IF-$\star$ case, though simple, is crucial, because it shows that our induction hypothesis is strong enough to guarantee that a comparison between two names in ρ will always have the same result as a comparison between two names in ρ'. In particular, the injectivity of the substitutions that make ρ and ρ' α-equivalent is necessary. □

This theorem leaves open the possibility that some α-variant of ρ might result in an environment ρ' that cannot yield a result. The following theorem says that if one α-variant terminates (either with a value or FAULT), then every α-variant can terminate (and, by Theorem 5.1, when it does, the value will be α-equivalent to the result of the original). [2]

Theorem 5.2 (α-equivalent environments have equivalent termination behavior)**.**

$$\begin{array}{ll} \text{If} & \tau = \text{typeof}(\Gamma, e) \\ & \text{and } \rho =_\alpha \rho' : \Gamma \\ & \text{and } \Gamma \vdash_{\text{exe}} \langle e, \rho \rangle \overset{k}{\Longrightarrow} w \\ \text{then} & \exists w'.\ \Gamma \vdash_{\text{exe}} \langle e, \rho' \rangle \overset{k}{\Longrightarrow} w' \text{ and } w =_\alpha w' : \tau \end{array}$$

Proof. (Sketch) For every choice of fresh name in the original computation, choose the same name in the other one. This preserves α-equivalence of ρ and ρ' for the induction hypothesis. □

5.1 Example

Consider again the example in Figure 2. Suppose that we had implemented a normal `let` construct (where the arms do *not* bind names from previous arms), with the type:

$$\begin{array}{rl} \textit{LetClauses} ::= & \text{Prod}() \\ & \mid \text{Prod}^{\Uparrow 1 \triangleright 0}(\text{Prod}^{\Uparrow 0}(\text{BAtom}, \textit{Expr}), \textit{LetClauses}) \end{array}$$

[2] Since we are using big-step semantics, we cannot talk directly about non-termination.

The only difference, besides the name, is that the recursive *LetClauses* does not have a $\downarrow 0$. If we had wanted to change the code in Figure 2 to expand ordinary `lets` instead, the above change to the type of *Expr* is sufficient, and the otherwise identical code would respect *LetClause*'s binding behavior! This is a consequence of Theorem 5.1, which ensures that programs cannot observe anything about names except their binding structure, as defined by their binding specifications.

6. Checking Binding Safety Statically

This section describes the Romeo deduction system. The purpose of this deduction system is to generate constraints (proof obligations) which, if satisfied, guarantee that escape, and therefore FAULT, will never occur (see Theorem 6.1).

The proof system's judgment is of the form $\Gamma \vdash_{\text{proof}} \{H\}\, e\, \{P\}$. Like the execution semantics, it is type-dependent, and for a similar reason: types control which atoms will be bound or free, and thus whether operations are valid or not.

We use H, P, and C to range over constraints. Typically, H, the hypothesis, contains facts (about the atoms in the environment) that are true by construction. P, the postcondition, contains predicates that describe the connection between atoms in the environment and atoms in the output. C is used for general constraints, and for the constraints in **let** statements.

The obligations emitted by the deduction system must be satisfied by showing them true for all ρ compatible with Γ; in practice, we do this with an SMT Solver (for example, Romeo-L [10] uses Z3 [9]).

We begin by giving the syntax of constraints.

$$\begin{array}{rcl} z \in \textit{ConstrSetVar} & ::= & x \\ & \mid & \cdot \\ s \in \textit{SetDesc} & ::= & \varnothing \\ & \mid & s \cup s \\ & \mid & s \cap s \\ & \mid & \mathit{sf}(z) \\ & \mid & \mathcal{F}_e(\Gamma) \\ \mathit{sf} \in \textit{SetFn} & ::= & \mathcal{F} \mid \mathcal{F}_r \mid \mathcal{F}_b \mid \mathcal{X} \\ H, P, C \in \textit{Constraint} & ::= & C \wedge C \\ & \mid & s = s \\ & \mid & s \neq s \\ & \mid & s \# s \\ & \mid & s \subseteq s \\ & \mid & z =_{\text{val}} e^{\text{qlit}} \\ & \mid & \textbf{true} \end{array}$$

Formulas are constructed from variables z, which range over program variables, and $\cdot$, which refers to the output value of the current expression. Set-valued terms are constructed from the free names ($\mathcal{F}$), free references ($\mathcal{F}_r$), free binders ($\mathcal{F}_b$), and exposable names ($\mathcal{X}$) of values, the free names of environments ($\mathcal{F}_e$), and then by the standard set constructors. Atomic formulas denote equality, inequality, etc., of sets, plus equality of values. Last, constraints are conjunctions of atomic formulas.

We use quasi-literals to describe values with variable interpolation. Because the type environment Γ is present, the type annotations of quasi-literals are redundant, but for economy of abstraction, we elected to reuse an existing concept instead of creating a new one.

In general, our rules are patterned after those in Pottier [13], using the type information in Γ to collect information about values. This subsumes Pottier's Δ. Most rules discharge their proof obligations by delegating them to proof obligations on subexpressions. The base cases of this recursion are P-CALL and P-QLIT, which describe proof obligations of the form $\Gamma \vDash H \Rightarrow P$. P-QLIT has only one obligation, which is to ensure that the result it produces

$$\frac{\text{typeof}\left(\Gamma, e^{\text{qlit}}\right) = \tau \qquad \Gamma, \cdot{:}\tau \vDash H \wedge \left(\cdot =_{\text{val}} e^{\text{qlit}}\right) \Rightarrow P}{\Gamma \vdash_{\text{proof}} \{H\}\ e^{\text{qlit}}\ \{P\}}\ \text{P-QLIT}$$

$$\frac{\begin{array}{c}\text{rettype}(f) = \tau \qquad \text{formals}(f) = (x_{\text{formal},i})_i \qquad \text{argtype}(f) = (\tau_{\text{formal},i})_i \\ \Gamma \vDash H \Rightarrow \text{pre}(f)\,[x_{\text{actual},i}/x_{\text{formal},i}]_i \qquad \Gamma, \cdot{:}\tau \vDash H \wedge \mathcal{F}(\cdot) \subseteq \mathcal{F}_{\text{e}}\left((x_{\text{actual},i}{:}\tau_{\text{formal},i})_i\right) \wedge \text{post}(f)\,[x_{\text{actual},i}/x_{\text{formal},i}]_i \Rightarrow P\end{array}}{\Gamma \vdash_{\text{proof}} \{H\}\ \left(f\ (x_{\text{actual},i})_i\right)\ \{P\}}\ \text{P-CALL}$$

$$\frac{x \text{ fresh for } \Gamma, H, P \qquad \Gamma, x{:}\text{BAtom} \vdash_{\text{proof}} \{H \wedge \mathcal{F}(x) \# \mathcal{F}_{\text{e}}(\Gamma)\}\ e\ \{P \wedge \mathcal{F}(x) \# \mathcal{F}(\cdot)\}}{\Gamma \vdash_{\text{proof}} \{H\}\ (\textbf{fresh}\ x\ \textbf{in}\ e)\ \{P\}}\ \text{P-FRESH}$$

$$\frac{\begin{array}{c}\forall i.\ x_i \text{ is fresh for } \Gamma, H, P \\ \Gamma(x_{\text{obj}}) = \text{Prod}_i^{\Uparrow \beta_{\text{e}}}(\tau_i{\downarrow}\beta_i) \qquad \Gamma, (x_i{:}\tau_i)_i \vdash_{\text{proof}} \left\{H \wedge \mathcal{X}(x_{\text{obj}}) \# \mathcal{F}_{\text{e}}(\Gamma) \wedge x_{\text{obj}} =_{\text{val}} \left(\textbf{prod}_i^{\Uparrow \beta_{\text{e}}}\ x_i{\downarrow}\beta_i\right)\right\}\ e\ \{P \wedge \mathcal{X}(x_{\text{obj}}) \# \mathcal{F}(\cdot)\}\end{array}}{\Gamma \vdash_{\text{proof}} \{H\}\ \left(\textbf{open}\ x_{\text{obj}}\ \left((x_i)_i\right)\ e\right)\ \{P\}}\ \text{P-OPEN}$$

$$\frac{\begin{array}{c}x \text{ fresh for } \Gamma, H, P, C \\ \Gamma \vdash_{\text{proof}} \{H\}\ e_{\text{val}}\ \{C\} \qquad \text{typeof}\,(\Gamma, e_{\text{val}}) = \tau_{\text{val}} \qquad \Gamma, x{:}\tau_{\text{val}} \vdash_{\text{proof}} \{H \wedge C\,[x/\cdot] \wedge \mathcal{F}(x) \subseteq \mathcal{F}_{\text{e}}(\Gamma)\}\ e_{\text{body}}\ \{P\}\end{array}}{\Gamma \vdash_{\text{proof}} \{H\}\ (\textbf{let}\ x\ \textbf{where}\ C\ \textbf{be}\ e_{\text{val}}\ \textbf{in}\ e_{\text{body}})\ \{P\}}\ \text{P-LET}$$

$$\frac{\Gamma(x) = \tau_0 + \tau_1 \qquad \Gamma, x_0{:}\tau_0 \vdash_{\text{proof}} \{H \wedge x =_{\text{val}} (\textbf{inj}_0\ x_0\ \tau_1)\}\ e_0\ \{P\} \qquad \Gamma, x_1{:}\tau_1 \vdash_{\text{proof}} \{H \wedge x =_{\text{val}} (\textbf{inj}_1\ \tau_0\ x_1)\}\ e_1\ \{P\}}{\Gamma \vdash_{\text{proof}} \{H\}\ (\textbf{case}\ x\ (x_0\ e_0)\ (x_1\ e_1))\ \{P\}}\ \text{P-CASE}$$

$$\frac{\Gamma \vdash_{\text{proof}} \{H \wedge \mathcal{F}(x_0) = \mathcal{F}(x_1)\}\ e_0\ \{P\} \qquad \Gamma \vdash_{\text{proof}} \{H \wedge \mathcal{F}(x_0) \# \mathcal{F}(x_1)\}\ e_1\ \{P\}}{\Gamma \vdash_{\text{proof}} \{H\}\ (\textbf{if}\ x_0\ \textbf{equals}\ x_1\ e_0\ e_1)\ \{P\}}\ \text{P-IFEQ}$$

$$\frac{(x_i{:}\tau_i)_i \vdash_{\text{proof}} \{C_0\}\ e\ \{C_1\}}{\vdash_{\text{proof}} \left(\textbf{define-fn}\left(f\ (x_i{:}\tau_i)_i\ \textbf{pre}\ C_0\right) : \tau_0\ e\ \textbf{post}\ C_1\right)\ \text{ok}}\ \text{P-FNDEF} \qquad \frac{\forall i.\ \vdash fD_i\ \text{ok} \qquad \epsilon \vdash_{\text{proof}} \{\textbf{true}\}\ e\ \{\textbf{true}\}}{\vdash_{\text{proof}} (fD_i)_i\ e\ \text{ok}}\ \text{P-PROG}$$

Figure 7. Verification rules for the deduction system

obeys whatever constraints were imposed in P, given that the environment satisfies the assumptions in H. P-CALL has two obligations; first, that the invoked function's precondition is true (given H), and second that the resulting value satisfies the constraints in P (given H and the postcondition of the function).

As one might expect, the key rules are P-FRESH and P-OPEN, whose definitions are closely connected to E-FRESH-OK and E-OPEN-OK.

Proving the theorems in Section 5 required our language to have two important properties: that (a) no name can escape the context that exposed it, except as a bound name, and (b) no name is exposed twice from two different binding relationships at the same time (thereby revealing equality of two names that are not related by binding). For the purposes of dynamically respecting α-equivalence, property (a) was enforced by detecting such a situation and emitting FAULT instead, and property (b) was established by the constraints imposed on the names exposed in E-FRESH-$\star$ and E-OPEN-$\star$.

Now, for the purposes of the deduction system, property (a) appears in the postcondition of both P-FRESH and P-OPEN, as an obligation to prove that the exposed free names are disjoint from the free names of the result value (spelled '$\cdot$'), because the purpose of the deduction system is to prevent FAULTs. On the other hand, property (b) is a guarantee provided by the language dynamics, and therefore appears in the hypothesis of both rules, saying that the exposed names are guaranteed to be disjoint from the environment so far.

The rules P-OPEN and P-CASE each add additional information to their hypotheses. This information conveys the relationship between the atoms in the scrutinee (x_{obj} and x_0 respectively) and the atoms in its component(s). In the P-CASE case, even though the underlying values are different, their sets of free atoms (and free binders and exposable atoms, etc.) are identical, so for the logic's purposes, they are equivalent.

6.1 Odds and ends

In the **let** expression, the body subexpression has the same result as the expression as a whole, but the value subexpression does not. Therefore, in P-LET, the condition C (whose $\cdot$ refers to the value subexpression) must be adjusted for use as a hypothesis for the body subexpression. Fortunately, the name x refers to the value subexpression in question, so a simple $[x/\cdot]$ substitution suffices. H may be used unchanged by both subexpressions because it will contain no references to $\cdot$.[3]

A similar issue occurs in P-CALL. The pre- and post-conditions of the function (not to be confused with the expression's postcondition P) are expressed relative to the formal parameters, which are meaningless out of context. Because the actual arguments to a function invocation are all required to be variable references (rather than allowing them to be whole subexpressions), the solution is again simple: a simultaneous substitution from the formals to the actuals suffices to make the pre- and post-conditions meaningful in the caller's context.

Shadowing amongst Romeo program variables is incompatible with the deduction system, because obligations must be able to refer to (and distinguish) everything in Γ by name. This gives rise to the requirement that certain x's be fresh for Γ, H, and P; this requirement is easily satisfied by a simple renaming pass prior to type and proof checking.

P-CALL's body hypothesis contains a term representing extra information as a consequence of Lemma 6.1, which states that the

[3] This is different from the strategy used in Pure FreshML [13], in which postconditions are functions that produce predicates.

free atoms in the result of any expression are a subset of the the free atoms in the environment in which it is evaluated. A similar term appears in the hypothesis for P-LET's body subexpression.The proofs of Lemmas 6.1 and 6.1, and the definition of the typesystem (including $\Gamma \vdash_{\text{type-env}} \rho$), can be found at `http://hdl.handle.net/2047/d20005013`.

Lemma 6.1 (No names made up)**.**

$$\begin{array}{ll} \text{If} & \tau = \text{typeof}(\Gamma, e) \\ & \text{and } \Gamma \vdash_{\text{type-env}} \rho \\ & \text{and } \Gamma \vdash_{\text{exe}} \langle e, \rho \rangle \overset{k}{\Longrightarrow} v \\ \text{then} & \text{fa}(\tau, v) \subseteq \text{fa}_{\text{env}}(\Gamma, \rho) \end{array}$$

Proof. By induction on k. □

Finally, in P-IFEQ, the result of the comparison can be expressed in our predicate language; in the branch in which the two atoms are equal, we note that their free atom sets (known to be singletons) are equal, and in the other branch, we note that their free atoms sets are disjoint.

6.2 Soundness of the Deduction System

The soundness of the deduction system is expressed in the following theorem.

Theorem 6.1 (Soundness of the deduction system)**.**

$$\begin{array}{ll} \text{If} & \tau = \text{typeof}(\Gamma, e) \\ & \text{and } \Gamma \vdash_{\text{type-env}} \rho \\ & \text{and } \Gamma \vdash_{\text{proof}} \{H\}\ e\ \{P\} \\ & \text{and } \Gamma \vdash_{\text{exe}} \langle e, \rho \rangle \overset{k}{\Longrightarrow} w \\ \text{then} & w \neq \text{FAULT} \text{ and } \Gamma, \cdot{:}\tau; \rho[\cdot \to w] \vDash H \Rightarrow P \end{array}$$

Proof. By induction on k. □

Observe that the theorem is unusually strong: it says that if there is *any* solution to the proof obligations, then the program is non-faulting for *any* suitably-typed environment ρ. This depends on the fact that the generation of proof obligations is deterministic (given pre- and post-conditions for the functions in the system).

6.3 Example

The Romeo-L code in Figure 2 contains a number of **open**s, each of which potentially can produce a FAULT. However, our deduction shows that FAULT will never happen. A complete derivation is too large to include here, but we will informally look at two examples.

First, on line 5, we are opening up a lambda abstraction. This "exposes" the lambda's binder (binding it to the variable *bv*). Fortunately, ($\mathbf{inj}_{\text{lambda}}$ (**prod** *bv*, *convert*(*e-body*) ↓0)), the body of the **open**, has no free names from its left-hand child and binds *bv* in its right-hand child, so (regardless of the output of *convert*) the exposed name from *bv* is not free in the result.

Second, on line 7, we open up the whole `let*` form, exposing all of the names that *lsc* exports. We must show that those names do not escape this context. When *lsc-some* is destructured, we know from its type that *bv* and *lsc-rest*, together, export that same set of names.

The value returned from the **open** is an application, constructed on line 13. Its left-hand-side is a lambda, which binds *bv* in *e-rest*. Therefore, we need to show that the names exported by *lsc-rest* are bound in *e-rest*. Fortunately, *e-rest* is a `let*` construct (line 12), defined to bind the names exported by *lsc-rest* in *e-body*, which is exactly what we needed. Therefore, the left-hand side of the the function application constructed by *convert* contains no free references that could cause a FAULT.

Now, we look at the right-hand-side of that application, which we generate by calling *convert*(*val-expr*). By Lemma 6.1, we know that *convert* produces a value whose free names are a subset of its argument. How do we know what names are free in *val-expr*? We know that, as an expression, it exports nothing, and so has no free binders. Any free references in it would have also been free in *lsc-some* (because it binds no names in the scope of its value expression), and therefore free in *let-star* itself. But *let-star* is part of the environment in which it was opened (on line 7), so, by the freshness of newly-exposed names, the names we are worried about must be fresh for *val-expr*.

A similar argument can be used to verify the safety of the other **open**s. In this example, the programmer didn't need to supply any constraints to justify the function calls. In general, constraints are necessary for the same reasons as in Pure FreshML [13], and the same examples apply.

7. Related work

7.1 Statically Specified Binding in Template Macros

The work of Herman and Wand [7, 8] introduced the idea of a static binding specification for a template or pattern-matching macro system (like Scheme's `syntax-rules`). Herman defined a language for binding specifications, and gave an algorithm for deciding whether a pattern-and-template macro was consistent with its binding specification. In practice, however, the complex macros in a language like Scheme are often not expressible in a pattern-matching system. Romeo provides a path for extending this macro system to a procedurally-based one, like Scheme's `syntax-case`.

Although our binding annotation system is very similar in power to Herman's, we have made some changes in representation. The most noticeable is that where Herman and Wand use addresses into binary trees of values, we use indices into wide products.

7.2 Pure FreshML

The second source for this work is Pure FreshML [13]. Both Pure FreshML and Romeo are first-order, side-effect-free languages in which a runtime system ensures that introduced names do not escape their scope, and both provide a proof system that generates proof obligations which, if true, guarantee statically that no faults will occur. One important difference, important for our intended application of macro-expanders, is that Romeo manipulates plain S-expression-like data, guided by types, whereas Pure FreshML saves type information in values. Our presentation of the language and semantics are somewhat different: for example, we have separate constructs for destructuring products (**open**) and destructuring sum types (**case**).

The system in [13] leaves the actual language of binding specifications underdetermined. All the formal development is done in a simple system, roughly equivalent to the λ-calculus, but one of the key examples, normalization by evaluation, is done using the more expressive system of Cαml [12]. Still, both of these systems are too weak to express complex binding constructs. For example, neither can express the natural syntax of the `let*` construct.

Also, our language provides stronger guarantees than does Pure Fresh ML. The primary claim in [13] is that no fresh name escapes its scope. This is much weaker than respecting α-equivalence. Consider a boolean-valued function that tests its arguments for syntactic (not α-) equivalence and returns a boolean. This function would not violate the no-escape condition, but of course it violates α-equivalence. We conjecture that Pure FreshML does respect α-equivalence, but we have not attempted to prove this.

7.3 Ott

Ott [16] is a system for metaprogramming that accepts binding specifications with a syntax and semantics similar to ours. However, Ott's goals are significantly different. Instead of providing a complete, name-aware programming system, Ott generates code for use in a theorem-prover, including definitions of types and a capture-avoiding substitution function. Ott supports a number of theorem-provers and a number of representations for the terms in them. Additionally, it can export boilerplate code for OCaml.

Ott's binding specifications are strictly more expressive than ours: effectively, they allow for a single value to export multiple sets of names (these sets are designated by "auxiliary functions"), which can be bound separately.

In order to support theorem-provers, Ott includes a definition for α-equivalence between its "concrete abstract syntax trees" (the equivalent of our values v). Their definition is based on a partial equivalence relation which relates two tree positions in a term if they are connected by binding (i.e., they would have to be renamed together). From this intra-tree relation, it is fairly straightforward to extract a notion of α-equivalence: two trees are α-equivalent if both (1) their free names match, and (2) the partial equivalence relations representing their bound names are identical. It is not clear whether this definition would lead to simpler proofs of Theorems like 5.1 and 5.2.

7.4 Hygienic Macro Systems in Scheme

The goals of our work have a great deal in common with the goals of hygienic macro systems, like those used in Scheme. There are two major problems in dealing with hygienic macro systems in Scheme. The first is that there is not yet a widely-accepted, formal, implementation-independent definition for the property of being hygienic. The algorithm described by Dybvig [3], which is the basis for hygienic macro expansion in Scheme, seems "correct" in the sense that it tends to behave consistently with the intuition of its users. But there is no specification against which it can be proved correct. The closest things to a specification, given by Clinger [2] and Dybvig [3], are phrased in terms of the bindings inserted or introduced by the macro. However, given a macro definition (say, in template style), there is no obvious way to tell what those bindings should be without reference to the expansion algorithm. So this specification remains circular. The use of static binding specifications (introduced for this purpose in [7]) provides a more rigorous basis for hygiene.

Secondly, the Dybvig algorithm offers no *static* guarantees. If the designer makes a mistake, it will only be discovered after the macro is expanded (and probably after it is used, perhaps by an innocent end-user). By contrast, our system offers a static guarantee: if the macro definition binds a name incorrectly, the error will be detected at macro-definition time by the deduction system, not as a runtime fault.

Even simple macros can cause unexpected results in traditionally-hygienic systems. Suppose we want a macro that translates

```
(lazy-let ((x (long-calculation)))
  ... x ...)
```

into

```
(let ((x (delay (long-calculation))))
  ... (force x) ...)
```

If the body of the `lazy-let` contains a macro invocation like `(my-macro1 x x)` how is the expander for `lazy-let` to know whether either of the `x`'s refers to the `x` in the declaration of the `lazy-let`? In the absence of a binding specification for `my-macro1`, the only solution is to expand the macros bottom-up, which is undesirable for other reasons. If we had a binding specification for `my-macro1`, we could perform this transformation without reference to the expansion of `my-macro1`.

7.5 Binding in Theorem-Proving Systems

Ever since the POPLMark Challenge [1], there has been a large interest in coding terms with bindings in various proof assistants [11][14][17]. These works have differing goals than ours; they are primarily concerned with proving facts about programs, while we are aiming at a usable meta-programming system. They also generally depend on representing abstract syntax trees in a pre-existing theorem-proving framework like Coq or Agda, whereas we are concerned with the complications of concrete syntax (even in an S-expression based language). We observe that Pouillard and Pottier [15] call a function "well-behaved" iff it preserves α-equivalence, and judge a type system to be satisfactory only if the functions definable in the system are well-behaved.

7.6 Extensions to Romeo

As we have described it, writing programs in Romeo is tedious. Programs must be written in an ANF-like style. For example, the arguments to function calls must be variables and not more complicated expressions.

A second problem is that we have not yet described how the truth of $\Gamma \vDash H \Rightarrow P$ is to be determined, and, if it fails, how the programmer is supposed to figure out how to fix it.

These problems are addressed in Muehlboeck's master's thesis [10], which presents a more usable front-end for Romeo, called Romeo-L. Romeo-L programs are written in a more natural dialect, and are automatically translated into the core Romeo we have described here. This translation introduces **let** expressions to avoid the need to program in A-normal form, and for each **let** it infers the strongest possible constraint C. Therefore the user need only supply constraints for function definitions. In practice, this means a vastly reduced annotation burden.

Romeo-L also includes a connection to the Z3 SMT solver [9], which is able to check statements of the form $\Gamma \vDash H \Rightarrow P$, completing the automated checking of the deduction system. Furthermore, it can translate counterexamples provided by Z3 into sets of names, so that the user can understand them, and it can explain how they violate a constraint either written by the user, or implicit in the rules for **fresh** or **open**. We hope to report on this in a separate paper.

Acknowledgments

We would like to thank Fabian Muehlboeck for his work in testing and validating Romeo, and J. Ian Johnson for his contributions to early versions of the binding system. We would also like to thank the anonymous reviewers for their comments to improve this paper.

This material is based on research sponsored by the Defense Advanced Research Projects Agency and the Air Force Research Laboratory. Any opinions, findings, conclusions or recommendations expressed herein are those of the authors, and do not necessarily reflect those of the US Government, DARPA, or the Air Force.

References

[1] B. E. Aydemir et al. Mechanized metatheory for the masses: The PoplMark challenge. In *Proceedings of the 18th International Conference on Theorem Proving in Higher Order Logics*, TPHOLs'05, pages 50–65, Berlin, Heidelberg, 2005. Springer-Verlag. ISBN 3-540-28372-2, 978-3-540-28372-0. URL `http://dx.doi.org/10.1007/11541868_4`.

[2] W. Clinger and J. Rees. Macros that work. In *Proceedings of the 18th ACM SIGPLAN-SIGACT Symposium on Principles*

of Programming Languages, POPL '91, pages 155–162, New York, NY, USA, 1991. ACM. ISBN 0-89791-419-8. URL http://doi.acm.org/10.1145/99583.99607.

[3] R. K. Dybvig, R. Hieb, and C. Bruggeman. Syntactic abstraction in scheme. *Lisp Symb. Comput.*, 5(4):295–326, Dec. 1992. ISSN 0892-4635. URL http://dx.doi.org/10.1007/BF01806308.

[4] S. Erdweg, T. van der Storm, and Y. Dai. Capture-avoiding and hygienic program transformations. In *Proceedings of the 28th European Conference on Object-Oriented Programming*. To appear.

[5] S. E. Ganz, A. Sabry, and W. Taha. Macros as multi-stage computations: type-safe, generative, binding macros in MacroML. *ACM SIGPLAN Notices*, 36(10):74–74–85–85, Oct. 2001. ISSN 0362-1340. URL http://portal.acm.org/citation.cfm?id=507669.507646.

[6] D. Hendriks and V. van Oostrom. Adbmal. In F. Baader, editor, *Automated Deduction – CADE-19*, volume 2741 of *Lecture Notes in Computer Science*, pages 136–150. Springer Berlin Heidelberg, 2003. ISBN 978-3-540-40559-7. URL http://dx.doi.org/10.1007/978-3-540-45085-6_11.

[7] D. Herman. *A Theory of Typed Hygienic Macros*. Ph.D. thesis, Northeastern University, 2010. URL http://www.ccs.neu.edu/home/dherman/research/papers/dissertation.pdf

[8] D. Herman and M. Wand. A theory of hygienic macros. In *Proceedings of the Theory and Practice of Software, 17th European Conference on Programming Languages and Systems*, ESOP'08/ETAPS'08, pages 48–62, Berlin, Heidelberg, 2008. Springer-Verlag. URL http://dx.doi.org/10.1007/978-3-540-78739-6_4.

[9] L. D. Moura and N. Bjørner. Z3: An efficient SMT solver. *Tools and Algorithms for the Construction and Analysis of Systems SE - 24*, 4963:337–340, 2008. URL http://dx.doi.org/10.1007/978-3-540-78800-3_24.

[10] F. Muehlboeck. *Checking Binding Hygiene Statically*. Master's thesis, Northeastern University, 2013. URL http://hdl.handle.net/2047/d20003134.

[11] R. Pollack, M. Sato, and W. Ricciotti. A canonical locally named representation of binding. *J. Autom. Reason.*, 49(2):185–207, Aug. 2012. ISSN 0168-7433. URL http://dx.doi.org/10.1007/s10817-011-9229-y.

[12] F. Pottier. An overview of Cαml. *Electronic Notes in Theoretical Computer Science*, 148(2):27–52, 2006. URL http://www.sciencedirect.com/science/article/pii/S1571066106001253.

[13] F. Pottier. Static name control for FreshML. In *Proceedings of the 22nd Annual IEEE Symposium on Logic in Computer Science*, LICS '07, pages 356–365, Washington, DC, USA, 2007. IEEE Computer Society. ISBN 0-7695-2908-9. URL http://dx.doi.org/10.1109/LICS.2007.44.

[14] N. Pouillard and F. Pottier. A fresh look at programming with names and binders. In *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming*, ICFP '10, pages 217–228, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-794-3. URL http://doi.acm.org/10.1145/1863543.1863575.

[15] N. Pouillard and F. Pottier. A unified treatment of syntax with binders. *Journal of Functional Programming*, 22(4–5):614–704, Sept. 2012. URL http://dx.doi.org/10.1017/S0956796812000251.

[16] P. Sewell et al. Ott: Effective tool support for the working semanticist. *J. Funct. Program.*, 20(1):71–122, Jan. 2010. ISSN 0956-7968. URL http://dx.doi.org/10.1017/S0956796809990293.

[17] C. Urban. Nominal techniques in Isabelle/HOL. *J. Autom. Reason.*, 40(4):327–356, May 2008. ISSN 0168-7433. URL http://dx.doi.org/10.1007/s10817-008-9097-2.