*JFP* **26**, e13, 52 pages, 2016.  
doi:10.1017/S0956796816000137

# *Romeo: A system for more flexible binding-safe programming**

PAUL STANSIFER and MITCHELL WAND

*College of Computer and Information Science,*
*Northeastern University Boston, Massachusetts, USA*
e-mail: pauls@ccs.neu.edu and wand@ccs.neu.edu

**Abstract**

Current systems for safely manipulating values containing names only support simple binding structures for those names. As a result, few tools exist to safely manipulate code 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 $\lambda_m$, but is a full-fledged binding-safe language like Pure FreshML.

## 1 Introduction

Name collision, the appearance of a name in a context where it means something other than what was intended, is usually thought of as a minor hazard of programming. But in the context of metaprogramming, the danger is significantly more serious, first because the metaprogrammer has no idea what names will appear in input syntax, and second because it is a natural programming practice to create copies of a single piece of syntax and expect each copy to behave independently, even when interleaved.

For example, suppose a metaprogrammer writes a function that, when given two lambda expressions, `(lambda (`$x_1$`) `$e_1$`)` and `(lambda (`$x_2$`) `$e_2$`)`, constructs

`(lambda (`$x_1$`) (lambda (`$x_2$`) (`$e_1$ $e_2$`))).`

If such a function were passed (syntax for) two identity functions, it should produce (syntax for) the composition function:

```
(lambda (a) (lambda (b) (a b)))
```

But if both its arguments are the same identity function, shadowing will (in many metaprogramming systems) change the meaning to something else:

```
(lambda (c) (lambda (c) (c c)))
```

* This material is based on research sponsored by the Defense Advanced Research Projects Agency and the Air Force Research Laboratory under agreement number FA8750-10-2-0233. 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.

Bugs resulting from name collision can be difficult to track down and fix. If the user of a metaprogramming system encounters such a bug, it will manifest, at best, as a static error in *generated* code. The user must reason about the internals of the metaprogramming system to determine how they caused that code to be generated. In the worst case, it will manifest as a behavioral anomaly that is invisible to code inspection.

Unfortunately, metaprogrammers do not typically receive much help from their languages and tools; a recent survey of nine tools for - domain-specific language implementation found that eight of them were vulnerable to errors or incorrect behavior as a result of name collision (Erdweg *et al*., 2014).

In order to solve this problem, it is necessary to codify the user's intentions. This is the responsibility of binding specifications, which indicate, for a particular term, how the set of names in scope in its subterms differ from the set of names in scope for it. Typically, each form in a language has a binding specification, and the binding behavior of a term is determined by looking up which form it corresponds to.

A system with binding specifications can be *binding-safe*. In binding-safe systems, $\alpha$-equivalent inputs proceed to $\alpha$-equivalent outputs (Herman & Wand, 2008). From the point of view of the user (either the programmer or the metaprogrammer), such a system behaves the same regardless of what names the user chooses. The primary benefit to the user is that name collision errors are impossible, but we have also discovered a secondary benefit: It is possible to write simpler code when terms can always be safely put into binding forms without fear of accidentally capturing free variables.

Two binding-safe metaprogramming systems are of particular interest to us: $\lambda_m$ and Pure FreshML.

### *1.1 The $\lambda_m$ calculus*

David Herman's $\lambda_m$ calculus (2010) is inspired by the world of Scheme macros.

Typically, Scheme is made up of a very small number of core forms, sufficiently expressive to write any program, but not user-friendly. For example, it might contain `lambda` but not `let`. The user (and the author of the standard library) uses metaprogramming to implement each user-friendly language form (macro) by describing how to translate (expand) it into core forms (or just simpler macros). The Scheme macro system orchestrates this expansion process.

Scheme predates the notion of binding safety, but Scheme programmers have been attacking the problem of name collision since at least 1986 (Kohlbecker *et al*., 1986). The standard approach Schemers have developed is called "hygiene". Unlike binding safety, which is a property of the metaprogramming system, hygiene is a property of the macro *expander*, operating by annotating names on the input and output of the metaprogram that is a macro implementation, rather than affecting the execution of the metaprogram itself. Furthermore, hygiene does not require binding specifications.

One disadvantage of this "lightweightness" is that, without binding specifications to indicate the intended binding behavior of macros, it is hard to say what it means

https://doi.org/10.1017/S0956796816000137 Published online by Cambridge University Press

for a system to be hygienic. In fact, is was not until 2015 that a notion of what it means for an implementation of hygiene to be correct was proposed (Adams, 2015). This makes it difficult to explain to the programmer how to understand hygiene, and makes it almost impossible to build any static guarantees off hygiene.

The $\lambda_m$-calculus addresses this by introducing binding specifications for macros, and then defining a binding-safe macro expander. One of the major challenges is that a single macro invocation may include arbitrarily many subterms, each with its own scope, and while each of those scopes may introduce a different set of names, the same name may be bound in many of them. $\lambda_m$'s binding specification system is powerful enough to express certain situations like this, including many standard Scheme macros.

The primary limitation of $\lambda_m$ (from our point of view) is that it is merely a pattern-matching macro system. This means that the metaprogrammer can only specify an expected invocation syntax for a macro (with "holes" for parameters), and a template for transcription (with interpolations from those holes), as opposed to writing a function that could perform arbitrary computation. In Scheme parlance, this is like `syntax-rules` (minus Macro By Example's treatment of `...`), as opposed to `syntax-case`. This makes many common macros impossible to define.

Furthermore, macros are only one of many forms of metaprogramming that can benefit from binding safety.

### *1.2 Pure FreshML*

Francois Pottier's Pure FreshML (2007a) is a core calculus of a functional language for metaprogramming, in which values are terms with binding structure. Programs in Pure FreshML are binding-safe. In contrast to systems that easily achieve binding safety by representing terms as functions (higher order abstract syntax) or by representing names canonically (de Bruijn indices), terms in Pure FreshML are ordinary trees that contain names, which seems to be the most ergonomic way to metaprogram.

The most important aspect of its enforcement of binding safety is that, whenever a binding term $v$ is destructured (i.e. passes through Pure FreshML's pattern-matching construct), the name it binds (say, $a$) is automatically "freshened", meaning that all occurrences of $a$ in $v$ are replaced by a new name. This prevents name collisions, but does not produce a fully binding-safe system.[1]

Binding safety requires $\alpha$-equivalent inputs to proceed to $\alpha$-equivalent outputs, which is essentially a guarantee of determinism. The generation of fresh names is non-deterministic, so Pure FreshML must ensure that the output of its programs don't contain any of those fresh names (at least as free names; non-determinism in bound names is acceptable according to the definition of binding safety). For a freshened name to escape the context, it was introduced in is an error, so Pure FreshML has a static proof system to prove that such an escape will never occur.

[1] Pure FreshML is based on FreshML (Shinwell *et al*., 2003), which stops here and achieves a less strong safety guarantee.

https://doi.org/10.1017/S0956796816000137 Published online by Cambridge University Press

| | | | |
|---|---|---|---|
| *CoreExpr* ::= | RAtom | (*variable reference*) | var-ref |
| | \| Prod(*CoreExpr*, *CoreExpr*) | (*application*) | apply |
| | \| Prod(BAtom, *CoreExpr*↓0) | (*abstraction*) | lambda |
| *Expr* ::= | RAtom | (*variable reference*) | — |
| | \| Prod(*Expr*, *Expr*) | (*application*) | — |
| | \| Prod(BAtom, *Expr*↓0) | (*abstraction*) | — |
| | \| Prod(*LetStarClauses*, *Expr*↓0) | (*sequential let*) | let* |
| *LetStarClauses* ::= | Prod() | (*no clauses*) | none |
| | $\mid \mathrm{Prod}^{\Uparrow 1 \triangleright 0}\begin{pmatrix}\mathrm{Prod}^{\Uparrow 0}(\mathrm{BAtom}, \mathit{Expr}),\\ \mathit{LetStarClauses}{\downarrow}0\end{pmatrix}$ | (*clause, and more clauses*) | lsc-some |

Fig. 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 in Figure 2.)

However, the only kind of binding form permitted in Pure FreshML is a pair of a name and a term, where the name is in scope for the term (in other words, the binding structure of a lambda term). This is partially addressed by Cαml (Pottier, 2006), which improves on Pure FreshML's expressivity somewhat. However, even Cαml's binding specifications are significantly limited. In particular, Cαml terms are divided into *expressions* (which contain references) and *patterns* (which contain binders), while terms in $\lambda_m$ can play both roles simultaneously.

### *1.3 Example*

For example, consider the following use of the `let*` syntactic form in Scheme, which exhibits a complex binding structure:

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

The `let*` 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, in the code above, all references to names are well-defined, and the value of `c` is 10. This behavior is similar to the behavior of `do` in Haskell and to telescopes in dependently typed languages.

We wish to "expand away" `let*`s. In Figure 1, we write some binding types using (a syntactically sugared version of) Romeo, 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. The α-equivalence-preserving nature of the translation is a property of Romeo, and the key is Romeo's destructuring construct, **open** (similar to the "`val <`$x_1$`>`$x_2$ `=` $e$" construct in FreshML (Shinwell *et al.*, 2003) or the **case** construct in Pure FreshML (Pottier, 2007b)). For example, (**open** *let-star* (*lsce-body*) ...) behaves the same way as the `(match let-star [(list lsc e-body) ...])` in Scheme, except that the value of *let-star* is α-converted in a way that avoids name collision (see E-Open-* in Section 4). This α-conversion is guided by the type of the value being destructured.

https://doi.org/10.1017/S0956796816000137 Published online by Cambridge University Press

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

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

If we had wanted to translate `let` (which only differs from `let*` in its binding behavior) instead, the exact same code would also serve that purpose; all that would need to be changed would be the types (see Section 4.3.2). We will discuss this example in more detail: We will cover the types in Section 2.1, the behavior of the code in Section 4.3.1, and the static guarantees the code provides in Section 8.3.

### *1.4 Contributions*

Our primary contribution is an extension of David Herman's system for binding-safety in a pattern-matching macro system (Herman, 2010) to cover macros defined by procedures, and thus general meta-programming for terms with bindings. Our language is inspired by Pure FreshML (Pottier, 2007b).

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 guarantees that instead of a name "escaping" the context in which it is defined, a FAULT is produced.
- Thanks to that guarantee, we prove a theorem showing that, in any execution, the dynamic environment can be replaced by one with $\alpha$-equivalent values, and that execution will proceed to a value $\alpha$-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

https://doi.org/10.1017/S0956796816000137 Published online by Cambridge University Press

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 $\mathit{CoreExpr}\downarrow 0$ 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 matching of related names together, the type for name introductions (or binders), BAtom, is made distinct from the type of names that reference binders, RAtom.

Instead of a binary product ($\times$), our type system has a "wide" product $(\mathrm{Prod}^{\Uparrow\beta_{ex}}\ (\tau_i\downarrow\beta_i)_i)$. Using a wide product increases the expressivity of our system because the position references on the right-hand-side of $\downarrow$ (and $\Uparrow$) are indices into the product. Manipulating forms that introduce multiple different but related scopes was one of the primary challenges in proving the correctness of our system.

Our system observes the convention that all names bound in a particular term are bound in all subterms, unless overridden by a new binding for the same name.[2]

Things get more complicated when the binders are exported longer distances up the tree. Returning to Figure 1, consider `let*`, in *Expr*. The "sequential let" line indicates that the binders exported by *LetStarClauses* are in scope in the body of the `let*` expression. The grammar for *LetStarClauses* says that a set of `let*`-clauses is either the empty list or the cons of a single clause and a *LetStarClauses*.

In the second production for *LetStarClauses*, the $\mathrm{Prod}^{\Uparrow 0}$ (BAtom, *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 $\mathit{LetStarClauses}\downarrow 0$ indicates that the names exported from the first clause are in scope in the remainder of the clauses.

The $\Uparrow (1 \triangleright 0)$ indicates that the entire *LetStarClauses* 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 $\Uparrow (1 \triangleright 0)$, except that duplicated atoms are an error.

Thus, our type system can be seen as an attribute grammar with a single attribute, whose values are sets of names representing bindings. These sets are synthesized from binders and values that export them, and inherited by every term underneath a term that imports one of them.

Our notations $\downarrow$, $\Uparrow$, $\triangleright$, 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] 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 (2003).)

https://doi.org/10.1017/S0956796816000137 Published online by Cambridge University Press

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

Fig. 3. Typed production rule for event handler.

### *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 at the time of the event. 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 (put-tag gui-elt (text-of kbd-evt)))
```

And, here is an $\alpha$-equivalent, but harder-to-read, version:

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

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

Regardless of whether they have the same names, the meanings of the two events must not be conflated, but in both subterms, the GUI element is the same. 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 user could have implemented the `handler` statement as a *function* with the following style of expected invocation:

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

If programmer convenience were irrelevant, languages would need no binding constructs other than lambda. 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.

https://doi.org/10.1017/S0956796816000137 Published online by Cambridge University Press

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

Values are either atoms, left- or right- injections of values (to model sum types), or tuples of values. We write $\mathbf{prod}\,(v_i)_i$ 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 $\mathrm{Prod}^{\Uparrow\beta_{\mathrm{ex}}}\,(\tau_0{\downarrow}\beta_0, \ldots,\ \tau_n{\downarrow}\beta_n)$, which we denote by the comprehension $\mathrm{Prod}^{\Uparrow\beta_{\mathrm{ex}}}\,(\tau_i{\downarrow}\beta_i)_i$, tells us how to interpret the value $\mathbf{prod}\,(v_i)_i$. The term $\beta_i$, constructed in our algebra of attributes, combines (a subset of) the binders exported by $v_0, \ldots, v_n$ to determine the local names bound in $v_i$. By convention, these names override those inherited from outside (above) $\mathbf{prod}\,(v_i)_i$. The binding combinator $\beta_{\mathrm{ex}}$, 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 whole tuple $\mathbf{prod}\,(v_i)_i$. If $\downarrow$ or $\Uparrow$ is omitted, the corresponding $\beta$ defaults to $\varnothing$.

Instead of parsing S-expressions, we have explicit 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 $\tau_0$ (resp. $\tau_1$).

Last, we have recursive types $\mu X.\tau$ (where $\tau$ must be productive), to interpret a value $v$ according to $\mu X.\tau$ is to interpret it according to $\tau\left[(\mu X.\tau)/X\right]$. To support it, we have type variables $X$, and type-level destructors $\mathbf{nth}_j$ . We define the latter as $\mathbf{nth}_j\ \mathrm{Prod}^{\Uparrow\beta_{\mathrm{ex}}}\,(\tau_i{\downarrow}\beta_i)_i \triangleq \tau_j$ (reusing our existing type constructor as a way to write type tuples, the $\beta_i$s and $\beta_{\mathrm{ex}}$ are ignored). The $\mathbf{nth}_j$ construct is uninteresting on its own, but it allows for the definition of mutually recursive types.

### 2.3 The algebra of binding combinators

Binding combinators are terms built from the following grammar:

$$
\begin{array}{rcl}
\ell & \in & \mathbb{N} \\
\beta \in \mathit{Beta} & ::= & \varnothing \mid \beta \uplus \beta \mid \beta \triangleright \beta \mid \ell.
\end{array}
$$

As discussed above, we use binding combinators to collect names from the sets exported by the subterms of a sequence $\mathbf{prod}\,(v_i)_i$. We will need to interpret these combinators as operating over both sets of names and substitutions (finite maps from names to names). As before, we make liberal use of comprehensions: We write

https://doi.org/10.1017/S0956796816000137 Published online by Cambridge University Press

$[\![\beta]\!]\,(A_i)_i$ for $[\![\beta]\!]\,(A_0, \ldots,\ A_n)$, etc. The interpretation is as follows:

$$\begin{aligned}
[\![_]\!]\,(_) \ &: \ \mathit{Beta} \times \overline{\mathit{AtomSet}} \to \mathit{AtomSet} \\
[\![\varnothing]\!]\,(A_i)_i &\triangleq \varnothing \\
[\![\ell]\!]\,(A_i)_i &\triangleq A_\ell \\
[\![\beta \triangleright \beta']\!]\,(A_i)_i &\triangleq [\![\beta]\!]\,(A_i)_i \cup [\![\beta']\!]\,(A_i)_i \\
[\![\beta \uplus \beta']\!]\,(A_i)_i &\triangleq [\![\beta]\!]\,(A_i)_i \uplus [\![\beta']\!]\,(A_i)_i\,.
\end{aligned}$$

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

In Romeo, constructing a value whose type contains a $\beta$ that contains a $\uplus$ that attempts to union two non-disjoint sets of names is an error. We omit checking for this error, as it is straightforward, and is merely provided for metaprogrammers to enforce their intended usage rules.

A substitution $\sigma$ is a partial function from atoms to atoms. For the purposes of manipulating them, we represent a substitution as a set of ordered pairs of atoms. Our substitutions are naive, which is to say that they ignore binding structure and simply affect all names. We interpret $\beta$'s on substitutions as follows:

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

We define the "override" operation $\sigma \triangleright \sigma'$ as follows:

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

The operation for combining disjoint substitutions $\sigma \uplus \sigma'$ is like $\triangleright$, except that it is undefinded if the domains of the substitutions in question overlap.

Now, using $[\![\beta]\!]\,(A_i)_i$, we can compute the *exported binders* from any value. These are also called the free binders, because they are considered to be free names in their term (the expected theorems relating free names to $\alpha$-equivalence only hold if we do this). The free binders of a value are determined using the value's type.

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

https://doi.org/10.1017/S0956796816000137 Published online by Cambridge University Press

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

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

The set of free atoms of a term is the union of the free references and the exported (or free) binders:

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

The set of exposable atoms is the set of those non-free names in a value that will become free when that value 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:

$$\mathrm{xa}\left(\mathrm{Prod}^{\Uparrow\beta_{\mathrm{ex}}}\,(\tau_i{\downarrow}\beta_i)_i\,,\mathbf{prod}\,(v_i)_i\right) \triangleq \left(\bigcup_i \mathrm{fb}(\tau_i,v_i)\right)\setminus \mathrm{fb}\left(\mathrm{Prod}^{\Uparrow\beta_{\mathrm{ex}}}\,\left(\tau_j{\downarrow}\beta_j\right)_j\,,\mathbf{prod}\,\left(v_j\right)_j\right)$$

Finally, we can determine whether a particular index is in the support of (i.e. referred to by) $\beta$:

$$\begin{aligned}
_\,\hat{\in}\,_ &\subseteq \mathbb{N}\times \mathit{Beta}\\
\ell\,\hat{\in}\,\varnothing &\triangleq \mathrm{false}\\
\ell\,\hat{\in}\,\ell' &\triangleq \left(\ell=\ell'\right)\\
\ell\,\hat{\in}\,\beta\triangleright\beta' &\triangleq \ell\,\hat{\in}\,\beta \text{ or } \ell\,\hat{\in}\,\beta'\\
\ell\,\hat{\in}\,\beta\uplus\beta' &\triangleq \ell\,\hat{\in}\,\beta \text{ or } \ell\,\hat{\in}\,\beta'
\end{aligned}$$

## 3 α-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 binders, and
- their non-exported binders can be renamed along with the names that reference them to make the terms identical.

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

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

https://doi.org/10.1017/S0956796816000137 Published online by Cambridge University Press

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

### *3.1 Binder equivalence*

Two values are $=_{\mathrm{B}}$ iff their exported (free) binders in the same positions are identical. Note that non-exported binders are irrelevant to $=_{\mathrm{B}}$, but are important to the calculation of $=_{\mathrm{R}}$.

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

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

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

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

For example,

$$\mathbf{prod}\,(a\ b\ c) =_{\mathrm{B}} \mathbf{prod}\,(a\ b\ d) : \mathrm{Prod}^{\Uparrow 0 \triangleright 1}\,(\mathrm{BAtom}, \mathrm{BAtom}, \mathrm{BAtom})$$

because non-exported atoms are ignored, but

$$\mathbf{prod}\,(a\ b\ c) \neq_{\mathrm{B}} \mathbf{prod}\,(b\ a\ c) : \mathrm{Prod}^{\Uparrow 0 \triangleright 1}\,(\mathrm{BAtom}, \mathrm{BAtom}, \mathrm{BAtom})$$

because exported names in each position must be the same.

### *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*

The first step in the wide product case is to match binders in identical positions with each other, for which we must define the $\bowtie$ operator (pronounced "join"). It walks through both values in lockstep, assigning a common fresh atom for each pair of corresponding binding atoms. The result is a pair of injective substitutions whose domains are equal to the set of exported binders (fb) of the values being joined.

To calculate $\bowtie$ in the wide product case, we recursively generate such a pair of substitutions for each subterm of the products being compared, make sure that the generated names (the ranges of those substitutions) are disjoint, and then combine those substitutions with $[\![\beta_{\mathrm{ex}}]\!]$. The resulting pair of substitutions is the output of the $\bowtie$ relation.

https://doi.org/10.1017/S0956796816000137 Published online by Cambridge University Press

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

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

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

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

Here and elsewhere, we use # to denote the disjointness operator over names, sets of names, and (the names in) values. It is naive, meaning that it entirely ignores binding structure. (Note that in J-Prod, we only examine the range of the substitutions without primes because the ranges of the two substitutions emitted by $\bowtie$ are identical.)

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 subterms 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 non-deterministically generate three names, which we will choose to be `aa`, `bb`, and `cc`. Then, we will have $\sigma_0 = \{\langle \mathtt{a}, \mathtt{aa}\rangle, \langle \mathtt{b}, \mathtt{bb}\rangle, \langle \mathtt{c}, \mathtt{cc}\rangle\}$ and $\sigma_0' = \{\langle \mathtt{d}, \mathtt{cc}\rangle\}$. The different ranges of these substitutions indicate that some names (the ones called `a` and `b` in the left-hand value) are shadowed and cannot be referred to at all by references on the right-hand side.

A more complete derivation is available in the supplementary material.

### 3.2.2 Comparison by substitution

Now, we can write the rules for $=_{\mathrm{R}}$. At the type RAtom, the atoms being compared are necessarily free, and are reference equal iff they are identical. Symmetrically to $=_{\mathrm{B}}$, any two atoms are $=_{\mathrm{R}}$ at BAtom.

At a wide product, the information from performing $\bowtie$ on the subterms pairwise is used to unify references that refer to binders in the same position. This is done as follows: For each pair of subterms $v_i$, $v_i'$, we use $\bowtie$ to generate a pair of substitutions $\sigma_i$, $\sigma_i'$ that rename the binders exported by $v_i$, $v_i'$ to be identical.

Then, for each subterm $v_i$, we compute $[\![\beta_i]\!]\,(\sigma_j)_j(v_i)$ (and the symmetrical value for $v_i'$), adjusting all the imported names (as defined in $\beta_i$) so that, regardless of the difference between $v_i$ and $v_i'$, references to binders in the same position become equal to a single new value (as generated by $\bowtie$).

https://doi.org/10.1017/S0956796816000137 Published online by Cambridge University Press

| $j$ | $v_j$ $v'_j$ | | $\sigma_j$ $\sigma'_j$ |
|---|---|---|---|
| 0 | `gui-elt` $\bowtie$ `a` | : BAtom → | $\{\langle$`gui-elt`,`gg`$\rangle\}$ $\bowtie$ $\{\langle$`a`,`gg`$\rangle\}$ |
| 1 | `mouse-evt` $\bowtie$ `b` | : BAtom → | $\{\langle$`mouse-evt`,`mm`$\rangle\}$ $\bowtie$ $\{\langle$`b`,`mm`$\rangle\}$ |
| 2 | `(handle-evt ...)` $\bowtie$ `(handle-evt ...)` | : *Expr* → | $\varnothing \bowtie \varnothing$ |
| 3 | `kbd-evt` $\bowtie$ `b` | : BAtom → | $\{\langle$`kbd-evt`,`kk`$\rangle\}$ $\bowtie$ $\{\langle$`b`,`kk`$\rangle\}$ |
| 4 | `(tag ...)` $\bowtie$ `(tag ...)` | : *Expr* → | $\varnothing \bowtie \varnothing$ |

Fig. 4. Substitutions generated for the handler example.

Note that this substitution is naive (that is, it disregards types and therefore binding). Even though we are only interested in the substitution's effect on free references, this naivete is acceptable because, first, $=_{\mathrm{R}}$ does not examine free binders, and second (broadly speaking), the substitution of unfree names is harmless (see Lemma 7.31). Because our substitutions are naive, we require that each substitution's range be disjoint from the values being examined (recall that $\bowtie$ non-deterministically chooses the substitutions' ranges). Without this requirement, we would have `(let* ((x 7)) x)` $=_{\mathrm{R}}$ `(let* ((y 7)) a)` : *Expr*, witnessed by $\sigma_0 = \{\langle \mathtt{x}, \mathtt{a}\rangle\}$ and $\sigma'_0 = \{\langle \mathtt{y}, \mathtt{a}\rangle\}$.

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

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

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

In our ongoing `let*` example, the appropriate substitution is a no-op on the *LetStarClauses*, which import nothing, but recursive application of $=_{\mathrm{R}}$ will discover their shared binding structure and compare them as equal. On the other hand, the *Expr* bodies will be both transformed into `(display cc)`, which lacks binding structure, and is naively equal to itself. So, the two expressions are $=_{\mathrm{R}}$. (They are also trivially $=_{\mathrm{B}}$, and therefore $=_{\alpha}$.) A complete derivation of their reference-equivalence, including analyzing the *LetStarClauses* themselves, is available in the supplementary material.

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 $\alpha$-equivalence, is in Figure 4.

Because the corresponding subterms import nothing, $\beta_0 = \beta_1 = \beta_3 = \varnothing$. $\beta_2$ is $0 \uplus 1$ and $\beta_4$ is $0 \uplus 3$. The result of performing those substitutions is shown in Figure 5, establishing the relationship between `mouse-evt` and the first `b` and the relationship between `kbd-evt` and the second `b`.

https://doi.org/10.1017/S0956796816000137 Published online by Cambridge University Press

$[\![0 \uplus 1]\!](\sigma_j)_j$((handle-evt gui-elt mouse-evt)) = (handle-evt gg mm)
$[\![0 \uplus 1]\!](\sigma'_j)_j$((handle-evt a b)) = (handle-evt gg mm)

$[\![0 \uplus 3]\!](\sigma_j)_j$((put-tag gui-elt (text-of kbd-evt))) = (put-tag gg (text-of kk))
$[\![0 \uplus 3]\!](\sigma'_j)_j$((put-tag a (text-of b))) = (put-tag gg (text-of kk))

Fig. 5. Result of substitution in the handler example (all other subterms are trivial).

## 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. We divide the task of achieving safety into three parts:

- 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, based on the non-escape property, we prove that at any point in execution, the dynamic environment could be replaced by one with $\alpha$-equivalent values, and execution would still proceed to a value $\alpha$-equivalent to what it otherwise would have. Execution is deterministic up to $\alpha$: that is, the non-deterministic choices that are made (e.g. for fresh identifiers) do not change the $\alpha$-equivalence class of the result.
- Last, we provide a deduction system (see Section 8) to generate proof obligations which, if satisfied, guarantee that escape (and thus, FAULT) will never occur.

The syntax of Romeo is given as follows:

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

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 8.

https://doi.org/10.1017/S0956796816000137 Published online by Cambridge University Press

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 name that is distinct from all other names in the execution environment. In order to use that name as a reference, the **ref** form takes an argument of type BAtom and returns it as a RAtom. A complete definition of the typechecking judgment can be found in the supplementary material.

We annotate injections with the types of the arm-not-taken, and product constructors with their binding structure. This allows us to synthesize the types of expressions with a function $\text{typeof}(\Gamma, e)$ whose definition is routine.

To simplify the deduction system, we require variables in some places where expressions would be more natural (like function arguments or $x_{\text{obj}}$ in **open**). As a result, programs are written in (roughly) A-normal form, naming intermediate results with **let**.

### *4.1 Operational semantics*

We define Romeo's execution in big-step style. An advantage, for our purposes, of the big-step style is that it allows us to simultaneously enforce constraints about the return values of and about the names generated by the **fresh** and **open** forms.

We begin with some auxiliary definitions that we will need

$$\begin{aligned} w \in \mathit{Result} &::= v \mid \textsc{fault} \\ \rho \in \mathit{ValEnv} &::= \epsilon \mid \rho\,[x \to v] \end{aligned}$$

$$\begin{aligned} \Gamma \in \mathit{TypeEnv} &::= \epsilon \mid \Gamma, x{:}\tau \\ \text{fa}_{\text{env}}(\Gamma, \rho) &= \bigcup_{x \in \text{dom}(\Gamma)} \text{fa}(\Gamma(x), \rho(x)) \end{aligned}$$

The form of the execution judgment is

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

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

### *4.2 Execution rules*

We can now give the rules for execution in Romeo. Rules that introduce names come in two forms, -Ok, and -Fail. In each case, the difference is that FAULT occurs in the -Fail 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.

Some execution rules will depend on the type environment $\Gamma$. This is because the binding structures of values are represented in their types ($\tau$), 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.

https://doi.org/10.1017/S0956796816000137 Published online by Cambridge University Press

$$\frac{a \notin \mathrm{fa}_{\mathrm{env}}(\Gamma,\rho) \qquad \Gamma, x{:}\mathrm{BAtom} \vdash_{\mathrm{exe}} \langle e, \rho\,[x \to a]\rangle \overset{k}{\Rightarrow} w \qquad \tau = \mathrm{typeof}\left((\Gamma, x{:}\mathrm{BAtom}), e\right) \qquad w = \textsc{fault} \vee a \notin \mathrm{fa}(\tau, w)}{\Gamma \vdash_{\mathrm{exe}} \langle (\textbf{fresh } x \textbf{ in } e), \rho\rangle \overset{k+1}{\Longrightarrow} w}\ \textsc{E-Fresh-Ok}$$

$$\frac{a \notin \mathrm{fa}_{\mathrm{env}}(\Gamma,\rho) \qquad \Gamma, x{:}\mathrm{BAtom} \vdash_{\mathrm{exe}} \langle e, \rho\,[x \to a]\rangle \overset{k}{\Rightarrow} w \qquad \tau = \mathrm{typeof}\left((\Gamma, x{:}\mathrm{BAtom}), e\right) \qquad w \neq \textsc{fault} \wedge a \in \mathrm{fa}(\tau, w)}{\Gamma \vdash_{\mathrm{exe}} \langle (\textbf{fresh } x \textbf{ in } e), \rho\rangle \overset{k+1}{\Longrightarrow} \textsc{fault}}\ \textsc{E-Fresh-Fail}$$

We begin with the rules for evaluating **fresh** expressions. The rules require that the new name not occur in the environment $\rho$. Our determinacy theorems (Theorems 7.1 and 7.2) guarantee that the choice of the new name will not affect the result (up to $\alpha$-equivalence). We have two versions of the rule: FRESH-FAIL, which returns FAULT when the new name appears free 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 convertible to a RAtom by the expression (**ref** $x$).

The hypothesis $\tau = \mathrm{typeof}\left((\Gamma, x{:}\mathrm{BAtom}), e\right)$ is needed to determine whether to produce FAULT or not. Determining the type, of course, is entirely static and could be pre-computed once rather than at each evaluation.

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

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

The next pair of rules destructure a product. Given a value $\rho(x_{\mathrm{obj}}) = \textbf{prod}\,(v_{\mathrm{obj},0}, \ldots, v_{\mathrm{obj},n})$, the **open** expression chooses an $\alpha$-variant **prod** $(v_0, \ldots, v_n)$ and binds the resulting pieces to the variables $x_i$. The free names in the $\alpha$-variant must be distinct both from names in the environment $\rho$ and from each other (to the extent that they are not actually related by binding). This is tested by a subsidiary judgment $\vdash_{\text{suff-disj}}$, discussed in Section 4.2.2. Chosing an $\alpha$-variant that satisfies this requirement is the subject of Section 5.

As with **fresh**, we have two rules which branch on whether any of the new names appear free 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 ($\alpha$-equivalence preservation) because the set of atoms free in the environment for $e$ but not in the environment for $\left(\textbf{open } x_{\mathrm{obj}}\ ((x_i)_i)\ e\right)$ (the atoms

https://doi.org/10.1017/S0956796816000137 Published online by Cambridge University Press

that "become free" in this execution step) is equal to the set $\mathrm{xa}\left(\tau_{\mathrm{obj}}, \mathbf{prod}\,(v_i)_i\right)$.

$$\frac{\Gamma \vdash_{\mathrm{exe}} \langle e_{\mathrm{val}}, \rho\rangle \overset{k_{\mathrm{val}}}{\Longrightarrow} v_{\mathrm{val}} \qquad \tau_{\mathrm{val}} = \mathrm{typeof}\,(\Gamma, e_{\mathrm{val}}) \qquad \Gamma, x{:}\tau_{\mathrm{val}} \vdash_{\mathrm{exe}} \langle e_{\mathrm{body}}, \rho\,[x \to v_{\mathrm{val}}]\rangle \overset{k_{\mathrm{body}}}{\Longrightarrow} w}{\Gamma \vdash_{\mathrm{exe}} \left\langle \left(\mathbf{let}\ x\ \mathbf{where}\ C\ \mathbf{be}\ e_{\mathrm{val}}\ \mathbf{in}\ e_{\mathrm{body}}\right), \rho\right\rangle \overset{k_{\mathrm{val}}+k_{\mathrm{body}}+1}{\Longrightarrow} w}\ \text{E-Let}$$

$$\frac{\Gamma \vdash_{\mathrm{exe}} \langle e_{\mathrm{val}}, \rho\rangle \overset{k}{\Rightarrow} \textsc{fault}}{\Gamma \vdash_{\mathrm{exe}} \left\langle \left(\mathbf{let}\ x\ \mathbf{where}\ C\ \mathbf{be}\ e_{\mathrm{val}}\ \mathbf{in}\ e_{\mathrm{body}}\right), \rho\right\rangle \overset{k+1}{\Longrightarrow} \textsc{fault}}\ \text{E-Let-Fail}$$

There are two evaluation rules for **let**, depending on whether calculating $e_{\mathrm{val}}$ faults. As noted above, the constraint $C$ is ignored at run-time.

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

As in Pure FreshML (Pottier, 2007b), 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.

$$\frac{\rho(x_{\mathrm{l}}) = a \qquad \rho(x_{\mathrm{r}}) = b \qquad a = b \qquad \Gamma \vdash_{\mathrm{exe}} \langle e_0, \rho\rangle \overset{k}{\Rightarrow} w}{\Gamma \vdash_{\mathrm{exe}} \langle (\mathbf{if}\ x_{\mathrm{l}}\ \mathbf{equals}\ x_{\mathrm{r}}\ e_0\ e_1), \rho\rangle \overset{k+1}{\Longrightarrow} w}\ \text{E-If-Yes}$$

$$\frac{\rho(x_{\mathrm{l}}) = a \qquad \rho(x_{\mathrm{r}}) = b \qquad a \neq b \qquad \Gamma \vdash_{\mathrm{exe}} \langle e_1, \rho\rangle \overset{k}{\Rightarrow} w}{\Gamma \vdash_{\mathrm{exe}} \langle (\mathbf{if}\ x_{\mathrm{l}}\ \mathbf{equals}\ x_{\mathrm{r}}\ e_0\ e_1), \rho\rangle \overset{k+1}{\Longrightarrow} w}\ \text{E-If-No}$$

$$\frac{\rho\left(x_{\mathrm{obj}}\right) = \mathbf{inj0}(v_0) \qquad \Gamma\left(x_{\mathrm{obj}}\right) = \tau_0 + \tau_1 \qquad \Gamma, x_0{:}\tau_0 \vdash_{\mathrm{exe}} \langle e_0, \rho\,[x_0 \to v_0]\rangle \overset{k}{\Rightarrow} w}{\Gamma \vdash_{\mathrm{exe}} \left\langle \left(\mathbf{case}\ x_{\mathrm{obj}}\ \ (x_0\ e_0)\ \ (x_1\ e_1)\right), \rho\right\rangle \overset{k+1}{\Longrightarrow} w}\ \text{E-Case-Left}$$

$$\frac{\rho\left(x_{\mathrm{obj}}\right) = \mathbf{inj1}(v_1) \qquad \Gamma\left(x_{\mathrm{obj}}\right) = \tau_0 + \tau_1 \qquad \Gamma, x_1{:}\tau_1 \vdash_{\mathrm{exe}} \langle e_1, \rho\,[x_1 \to v_1]\rangle \overset{k}{\Rightarrow} w}{\Gamma \vdash_{\mathrm{exe}} \left\langle \left(\mathbf{case}\ x_{\mathrm{obj}}\ \ (x_0\ e_0)\ \ (x_1\ e_1)\right), \rho\right\rangle \overset{k+1}{\Longrightarrow} w}\ \text{E-Case-Right}$$

The remainder of the rules are routine. For simplicity's sake, the equality test construct works only on atoms. But in Romeo, for any type, it is straightforward to write an equality test out of **equals**, and that predicate will necessarily be an $\alpha$-equivalence.

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

https://doi.org/10.1017/S0956796816000137 Published online by Cambridge University Press

The E-Prog rule initiates evaluation of a program. It is notionally responsible for setting up the function context, which we omit from our notation, as it is otherwise constant.

### 4.2.1 Quasi-Literals

The last kind of Romeo expression is the *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 make type synthesis possible.

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

Their evaluation is routine, and is specified by the following rules:

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

### 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 from before

```
(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 each destructured without renaming, we would have `d` = `d` = `d`, even though `a` ≠ `b` ≠ `c`, violating our goal of being indifferent to α-conversion. Therefore, E-Open potentially needs to freshen each binder to a distinct new name, e.g.

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

The rule to ensure this is that, before destructuring, we must α-convert values so that the binders exposed by destructuring are disjoint from each other and from

https://doi.org/10.1017/S0956796816000137 Published online by Cambridge University Press

any names that appear in the environment. This gives rise to the hypothesis

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

in the E-OPEN-$\star$ rules.

To calculate $\vdash_{\text{suff-disj}}$, we need the judgment $\vdash_{\text{bndrs-disj}} v : \tau$, which checks that the exported binders in $v$ (as determined by the type $\tau$) are disjoint from each other.

$$\frac{}{\vdash_{\text{bndrs-disj}} a : \mathrm{BAtom}}\ \text{BD-BATOM} \qquad \frac{}{\vdash_{\text{bndrs-disj}} a : \mathrm{RAtom}}\ \text{BD-RATOM}$$

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

We can now define $\vdash_{\text{suff-disj}}$, which checks the disjointness of *non*-exported subterms (because these are the binders that will become free after destructuring), and also that those names (calculable by xa) are disjoint from a set $A$ of atoms (in practice, this is the set of free atoms in the environment).

$$\frac{\forall i \mathrel{\hat{\notin}} \beta_{\mathrm{ex}}.\ \vdash_{\text{bndrs-disj}} v_i : \tau_i \qquad \forall i, j \mathrel{\hat{\notin}} \beta_{\mathrm{ex}}.\ i \neq j \Rightarrow \mathrm{fb}(\tau_i, v_i) \mathbin{\#} \mathrm{fb}(\tau_j, v_j) \qquad \mathrm{xa}\big(\mathbf{prod}\,(v_i)_i, \mathrm{Prod}^{\hat{\Uparrow}\beta_{\mathrm{ex}}}\,(\tau_i{\downarrow}\beta_i)_i\big) \mathbin{\#} A}{A \vdash_{\text{suff-disj}} \mathbf{prod}\,(v_i)_i : \mathrm{Prod}^{\hat{\Uparrow}\beta_{\mathrm{ex}}}\,(\tau_i{\downarrow}\beta_i)_i}\ \text{SUFF-DISJ}$$

## *4.3 Examples*

### *4.3.1 Translation of* `let*`

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 (Muehlboeck, 2013), 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 constraint $C$ of **let**, so we may omit it. Furthermore, it will turn out (see Section 8.3) that our example needs no pre- or post-conditions from *convert* to show the absence of FAULT, so those constraints are also omitted.

Additionally, we have written our code using more readable $n$-way sum types. This means that our **case** construct can branch four 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. In the case where `let*` has at least one arm, line 12 constructs a smaller `let*` with one fewer arm, and recursively converts it, calling

https://doi.org/10.1017/S0956796816000137 Published online by Cambridge University Press

the result *e-rest*. Finally, lines 13–14 construct a beta-redex in the object language to bind the first arm's name to its value expression in *e-rest*.

#### 4.3.2 Translation of `let`

Consider again the example above. Suppose that we had implemented a normal `let` construct (where names from previous arms are not in scope for the later arms), with the type:

$$\begin{aligned} \mathit{LetClauses} ::= &\ \mathrm{Prod}\,() \\ &\mid \mathrm{Prod}^{\Uparrow 1 \triangleright 0}\left(\mathrm{Prod}^{\Uparrow 0}\left(\mathrm{BAtom}, \mathit{Expr}\right), \mathit{LetClauses}\right) \end{aligned}$$

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 `let`s instead, the above change to the type of *Expr* is sufficient, and the otherwise identical code would respect *LetClause*'s binding behavior and correctly expand the `let` construct! This is a consequence of Theorem 7.1, which ensures that programs cannot observe anything about names except their binding structure, as defined by their binding specifications.

## 5 Freshening

Executing the E-Open-* rules in a Romeo implementation requires the ability to take an arbitrary product value $\mathbf{prod}\,(v_i)_i$, and generate a new $\alpha$-equivalent value $\mathbf{prod}\,\left(v_i'\right)_i$ such that $A \vdash_{\text{suff-disj}} \mathbf{prod}\,\left(v_i'\right)_i : \tau$ (for a particular $A$). We call this process "freshening".

### 5.1 Approach

Conceptually, the $\vdash_{\text{suff-disj}}$ predicate (motivated and defined in Section 4.2.2) requires that all exposable binders in a value be mutually disjoint (and disjoint from $A$). This is easy to achieve: simply assign fresh names to each exposable binder. However, in order to produce a value that is also $\alpha$-equivalent to the input, we must also rename all references that refer to those values.

The difficulty in this is best illustrated by an example. Suppose that we have started freshening, and we've chosen to rename the binder `y` to `yy` in the following binding form, and thus, to maintain $\alpha$-equivalence, must rename all of its references:

```
(let* ((x (string-length y))
       (y 5)
       (z (+ y 1)))
  (+ y 2))
```

Because the binder `y` is both imported (into `(z (+ y 1))`) and exported (out to the `let*` and then imported into `(+ y 2)`), the substitution $\{\langle \mathtt{y}, \mathtt{yy}\rangle\}$ must be applied to multiple places. However, applying the substitution to the whole value is unacceptable because the free reference `y` in `(string-length y)` must *not* be renamed.

https://doi.org/10.1017/S0956796816000137 Published online by Cambridge University Press

Therefore, after generating a fresh replacement for y, we must use ↓ (import) designations in the type of the value to determine where that particular y is in scope, and only rename references to y in those locations, in order to keep references in sync with the changes we made to the binders.

### *5.2 Reference renaming*

While the freshening operation only affects *non-free* names in the value we are freshening, we will accomplish it by renaming the *free* names of its subterms (and sub-subterms, etc.). This is because those are the non-free names whose meaning is determined by imports from other subterms. To that end, we need an operation that applies a renaming only to the free references of a term. It needs to know the type of its value argument so that it can avoid touching binders and bound names. With that information, the implementation is straightforward.

$$
\begin{aligned}
{-|_{\mathrm{r}}(-:-)} &: \mathit{Subst} \times \mathit{Value} \times \mathit{Type} \to \mathit{Value} \\
\sigma|_{\mathrm{r}}(a{:}\mathrm{BAtom}) &\triangleq a \\
\sigma|_{\mathrm{r}}(a{:}\mathrm{RAtom}) &\triangleq \sigma(a) \\
\sigma|_{\mathrm{r}}\left(\mathbf{prod}\,(v_i)_i : \mathrm{Prod}^{\Uparrow\beta_{\mathrm{ex}}}\,(\tau_i{\downarrow}\beta_i)_i\right) &\triangleq \mathbf{prod}\left(\left(\sigma \setminus A_i\right)\big|_{\mathrm{r}}(v_i{:}\tau_i)\right)_i \\
&\quad \text{where } \forall i.\ A_i \triangleq [\![\beta_i]\!]\left(\mathrm{fb}(v_j, \tau_j)\right)_j
\end{aligned}
$$

### *5.3 Implementation of freshening*

Let $v_{\mathrm{top}}$ denote the term we are freshening, and $v'_{\mathrm{top}}$ denote the result of freshening.

We begin by defining freshen-subterm, which performs freshening on a subterm of $v_{\mathrm{top}}$. It takes a value $v$ (which is assumed to be a subterm of $v_{\mathrm{top}}$), the type $\tau$ of that value, and a boolean parameter, *exported-from-v-top?*, which is true iff the exported binders from $v$ are exported in an unbroken chain all the way out of $v_{\mathrm{top}}$.

The function freshen-subterm applied to $v$ returns a value $v'$ and a substitution $\sigma$ such that:

- $A \vdash_{\text{suff-disj}} v'{:}\tau$,[3]
- $v =_{\mathrm{R}} v' : \tau$ (i.e. the structure of bound atoms and the free references are unchanged),
- $\sigma(v) =_{\mathrm{B}} v' : \tau$ (i.e. the exported binders differ by $\sigma$),
- $\mathrm{dom}(\sigma) = \mathrm{fb}(v, \tau)$,[4] and
- if *exported-from-v-top?* is true, $v =_{\mathrm{B}} v' : \tau$ (i.e. the exported binders are identical, and thus $v =_{\alpha} v' : \tau$ and $\sigma$ is a no-op),

[3] For the sake of correctness, it is only necessary for $v'_{\mathrm{top}}$ (not all of its subterms, sub-subterms, etc.) to be sufficiently-disjoint. At the cost of implementation complexity, it is possible to optimize this algorithm so that it only freshens exposable names.

[4] This means that, even when $\sigma$ is a no-op substitution, it is still not an empty set. This is because of shadowing: If an import brought a particular name in twice, with a non-freshened version of it shadowing a freshened version of it, it is necessary that the non-freshened version correctly override the freshened version when the substitutions are combined by $[\![]\!]$.

https://doi.org/10.1017/S0956796816000137 Published online by Cambridge University Press

- but if *exported-from-v-top?* is false, the exported binders of $v'$ are fresh.

We omit the mechanics of threading environments of names through freshen-subterm, as they are routine. Informally speaking, "where *aa* is fresh" means that all generated names must be distinct from $A$ and from each other.

The base cases of our recursive freshening function are all straightforward. References do not require any action at this level, but binders are freshened, provided they are not exported from $v_{\text{top}}$. This ensures that $v_{\text{top}} =_{\text{B}} v'_{\text{top}} : \tau_{\text{top}}$.

Either way, we also return a substitution that reflects the change (if any) in the exported binders of the result.

$$\begin{aligned}
&\text{freshen-subterm}\,(_:_,_) \quad : \quad \mathit{Value} \times \mathit{Type} \times \mathit{Bool} \to \mathit{Value} \times \mathit{Subst} \\
&\text{freshen-subterm}\,(a{:}\text{RAtom},_) \triangleq \quad a, \varnothing \\
&\text{freshen-subterm}\,(a{:}\text{BAtom},\mathbf{true}) \triangleq \quad a, \{\langle a, a\rangle\} \\
&\text{freshen-subterm}\,(a{:}\text{BAtom},\mathbf{false}) \triangleq \quad aa, \{\langle a, aa\rangle\}\text{, where } aa \text{ is fresh}
\end{aligned}$$

Given a wide product $\mathbf{prod}\,(v_i)_i$, we first recursively call freshen-subterm on all of the subterms $v_i$, producing values $v'_i$ whose binders have been renamed according to $\sigma_i$. Observe that $v_i$ exports from $v_{\text{top}}$ iff $\mathbf{prod}\,(v_i)_i$ does and $v_{\text{top}}$ exports $v_i$.

Then, everywhere subterm $j$ is imported, we rename free references according to $\sigma_j$. We do this by calculating, for each subterm $i$, the value $[\![\beta_i]\!]\,(\sigma_j)_j$.

Finally, we have to determine what substitution corresponds to the difference in exported binders between $\mathbf{prod}\,(v_i)_i$ and the value we return. This is calculated by $[\![\beta_{\text{ex}}]\!]\,(\sigma_j)_j$.

$$\begin{aligned}
&\text{freshen-subterm}\left(\mathbf{prod}\,(v_i)_i : \text{Prod}^{\Uparrow\beta_{\text{ex}}}\,(\tau_i{\downarrow}\beta_i)_i, \mathit{exported\text{-}from\text{-}v\text{-}top?}\right) \\
&\quad \triangleq \mathbf{prod}\left(\left[\!\left[\beta_i\right]\!\right]\,(\sigma_j)_j\Big|_{\text{r}}\,(v'_i{:}\tau_i)\right)_i, \quad \left[\!\left[\beta_{\text{ex}}\right]\!\right]\,(\sigma_j)_j \\
&\qquad \text{where } \forall j.\ \left(v'_j, \sigma_j\right) \triangleq \text{freshen-subterm}\left(v_j{:}\tau_j, \mathit{exported\text{-}from\text{-}v\text{-}top?} \wedge \left(i \mathrel{\hat{\in}} \beta_{\text{ex}}\right)\right)
\end{aligned}$$

Using freshen-subterm to turn $v_{\text{top}}$ to $v'_{\text{top}}$ is now simple. We generate a sufficiently disjoint yet $\alpha$-equivalent value to $v_{\text{top}}$ by using freshen-subterm with *exported-from-v-top?* set to **true**, as everything exported from $v_{\text{top}}$ is (tautologically) exported from $v_{\text{top}}$. We discard the resulting substitution (but by the invariants above, we know it to be a no-op whose domain is $\text{fb}\left(v_{\text{top}}, \tau_{\text{top}}\right)$).

$$\begin{aligned}
&\text{freshen}\,(_:_) \quad : \quad \mathit{Value} \times \mathit{Type} \to \mathit{Value} \\
&\text{freshen}\left(v_{\text{top}}{:}\tau_{\text{top}}\right) \triangleq v'_{\text{top}} \\
&\qquad\qquad \text{where } v'_{\text{top}}, _ \triangleq \text{freshen-subterm}\left(v_{\text{top}}{:}\tau_{\text{top}}, \mathbf{true}\right)
\end{aligned}$$

Thus, by the (conjectured) invariants above (and recalling that the informal treatment of freshness means that freshening doesn't explicitly examine $A$), we have:

*Conjecturea 5.1* (*Freshening is always possible*)
If $v' = \text{freshen}\,(v{:}\tau)$, then $v =_\alpha v' : \tau$ and $A \vdash_{\text{suff-disj}} v'{:}\tau$.

This shows that it is possible to construct a value $\mathbf{prod}\,(v_i)_i$ that satisfies the $\vdash_{\text{suff-disj}}$ and $=_\alpha$ premises of E-Open-$\star$, and thus, it is possible to execute Romeo programs.

https://doi.org/10.1017/S0956796816000137 Published online by Cambridge University Press

## 6 Romeo respects α-equivalence: a guide to the proofs

We are now ready to prove our main theorem: that Romeo respects α-equivalence. Romeo is non-deterministic in its choice of names in the E-FRESH-⋆ and E-OPEN-⋆ rules. This complicates the proof of respecting α-equivalence. In a system with simpler binding structures, like Pure FreshML, the proof would go as follows:

- Define all the states of the machine in Section 4 to be α-equivalence classes.
- Rely on the freshness condition for binders (Pouillard & Pottier, 2010) to show that each manipulation on machine states (defined in terms of α-equivalence-class representatives) respects α-equivalence.

In such a system, we would have needed to prove only Lemma 8.1, as does Pottier (2007a).

Unfortunately, we were unable to usefully model our complex binding structures, especially buried bindings, in nominal logic. Therefore, we were forced to proceed from first principles.

We show two results: First, that if the evaluations of an expression in two α-equivalent environments both terminate, then their results are α-equivalent, and second, if the evaluation of an expression in one of two α-equivalent environments yields a result, then evaluating the expression in the other one must yield at least one α-equivalent result as well. Since we are using big-step semantics, we cannot talk directly about non-termination.

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

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 in the supplementary material. Here, we discuss the most interesting proofs. It is typical for reasoning of this complexity to be done in a mechanical reasoning system like Coq, but because we initially underestimated its size and scope, our proof is entirely handwritten.

### *6.1 Executions on α-equivalent environments yield α-equivalent results*

The first, and hardest, part of proving Romeo's soundness is Theorem 7.1, which shows that two terminating executions of α-equivalent environments yield α-equivalent result values.

The major problem in the proof is that the two executions will potentially generate different fresh names in E-FRESH-⋆ and E-OPEN-⋆. Hence, even though the environments start out α-equivalent, they will not stay α-equivalent throughout execution. For example, a **fresh** statement non-deterministically introduces a new name into the environment. Therefore, we must generalize our induction hypothesis to account for the ways in which $\rho$ and $\rho'$ diverge from α-equivalence.

We account for this divergence with two injective substitutions that unify the names the two executions introduce. So our induction hypothesis says that if $\sigma \circ \rho =_\alpha$

https://doi.org/10.1017/S0956796816000137 Published online by Cambridge University Press

$\sigma' \circ \rho' : \Gamma$ for a pair of injective substitutions $\sigma$ and $\sigma'$, then the results will be $\alpha$-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, $\sigma$ and $\sigma'$ are extended to map the new names to a common fresh name (lines 7.49–7.50 in the proof in Section 7.3), for the sake of the induction hypothesis.

The induction hypothesis tells us that the results (of the recursive evaluation of $e$) are equivalent modulo the *extended* substitutions (line 7.55). The result of the **fresh** evaluation step is either the same as that of the recursive step, or FAULT. We first show that the *original* substitutions suffice to $\alpha$-equate those two values (lines 7.61–7.62), and then that one side faults if and only if the other side does (lines 7.57–7.58).

The E-OPEN-$\star$ case proceeds with a similar structure.[5] 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 (Lemma 7.42, used on lines 7.74–7.77, described in Section 6.1.1) states that the $\vdash_{\text{suff-disj}}$ predicate in Romeo's execution rules is strong enough that the technique from the E-FRESH-$\star$ rules works for **open**, even though the various subterms of the value being destructured have potentially different scopes. In a sense, the lemma is where the complex binding structures meet the binding-safe programming in our system, and it is discussed in more detail in Section 6.1.1. 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 $\rho$ will always have the same result as a comparison between two names in $\rho'$. This is where the injectivity of the substitutions $\sigma$ and $\sigma'$ is used.

### *6.1.1 One substitution, not one per subterm, suffices for opening*

Lemma 7.42 is a crucial part of our project to extend binding-safe programming to support complex binding structures. A binding form in our system may possess many scopes, with different meanings for the same names. However, to programmatically manipulate such a binding form, it must be destructured, which dumps all of those differently scoped subterms into the same execution environment.

This lemma is our most interesting technical trick, showing that, if $A \vdash_{\text{suff-disj}} \mathbf{prod}\,(v_i)_i : \tau$ holds (where $A$ is the set of atoms in the environment), destructuring is safe. For the purposes of Theorem 7.1, "destructuring is safe" means that a single pair of substitutions suffices to bring all the subterms "back" into $\alpha$-equivalence (in other words, it will not lose information by causing a name collision).

[5] In fact, **open** can be used to replace **fresh** (whenever at least one binding form exists in the environment). We keep both constructs because we found it easiest to understand the E-OPEN-$\star$ cases of our proofs a generalization of the E-FRESH-$\star$ cases.

https://doi.org/10.1017/S0956796816000137 Published online by Cambridge University Press

The proof of this lemma starts on line 7.13, which generates, not a single pair of substitutions, but one pair for each subterm of the pair of values. Lines 7.16–7.17 adjust those substitutions to avoid free names in the environment, and lines 7.18–7.26 restrict those substitutions by removing any exported names, and show that the resulting substitutions (interpreted by $\beta_i$) still can achieve $=_R$ of pairs of subterms of the values.

Lines 7.27–7.29 show that all those substitutions can be combined to form a single pair of substitutions. Examining one side of that pair, lines 7.30–7.37 show that that single substitution is an adequate replacement for any of the substitutions generated on line 7.22.

Lines 7.40–7.41 show that, for exported subterms, their exclusion from the substitutions means that they remain $=_B$ even after being substituted.

Lines 7.42–7.43 show that, for subterms that are not exported, the overall pair of substitutions makes them $=_B$.

### *6.1.2 α-equivalence and free names are connected*

Lemma 7.30 (along with its cases Lemmas 7.28 and 7.29, and its corollary, Lemma 7.31) is used extensively in our proof, as it links the concepts of free names and α-equivalence. It states that a name is free if and only if a (naive) substitution of that name leads to a non-α-equivalent value.

The case for binders (Lemma 7.29) is straightforward, but the case for references (Lemma 7.28) is much more complex. Its sublemma, lines 7.1–7.6, applies the induction hypothesis in the case where the name being substituted is not imported by the overall wide product in the current subterm. The rest of it is a case analysis between (a) *a* being a free reference in the value, and therefore also a free reference in one of its subterms (lines 7.7–7.8) , (b) *a* being non-free, which turns into a case analysis for each subterm: (b.i) *a* not being imported (lines 7.11–7.12), and (b.ii) *a* being imported. This final sub-case does not use the sublemma, but instead shows that the substitutions we generated in line 7.10 prove reference-equivalence of the whole value.

## *6.2 Termination is representation-oblivious*

Theorem 7.1 leaves open the possibility that for a $\rho$ whose execution leads to a result *w*, execution of some α-variant $\rho'$ on the same program might fail to terminate. However, Theorem 7.2 says that if one α-variant terminates (either with a value or FAULT), then the execution of every α-variant will terminate (and, by Theorem 7.1, when it does, the value will be α-equivalent to the result of the original). Recall that we cannot talk directly about non-termination in our big-step semantics.

Broadly speaking, our approach to proving Theorem 7.2 is that, for every choice of fresh name in the original computation, we choose the same name in the other one. This preserves α-equivalence of $\rho$ and $\rho'$ for the induction hypothesis.

https://doi.org/10.1017/S0956796816000137 Published online by Cambridge University Press

## 7 Soundness of execution for α-equivalence: lemmas and key proofs

### 7.1 General definitions

#### 7.1.1 Naive operations

First, we show some basic properties regarding our naïve operations. These follow from straightforward inductive arguments.

*Lemma 7.1* (*Substitutions never increase disjointness*)
If $\sigma(A) \# \sigma(A')$, then $A \# A'$.

$$\begin{aligned} \mathrm{supp}(\_) &: \mathit{Value} \to \mathit{AtomSet} \\ \mathrm{supp}(a) &\triangleq \{a\} \\ \mathrm{supp}(\mathbf{inj0}(v)) &\triangleq \mathrm{supp}(v) \\ \mathrm{supp}(\mathbf{inj1}(v)) &\triangleq \mathrm{supp}(v) \\ \mathrm{supp}(\mathbf{prod}\,(v_i)_i) &\triangleq \bigcup_i \mathrm{supp}(v_i) \end{aligned} \qquad \begin{aligned} \mathrm{supp}(\_) &: \mathit{Subst} \to \mathit{AtomSet} \\ \mathrm{supp}(\sigma) &\triangleq \mathrm{dom}(\sigma) \cup \mathrm{rng}(\sigma) \\ \\ \mathrm{supp}(\_) &: \mathit{ValEnv} \to \mathit{AtomSet} \\ \mathrm{supp}(\rho) &\triangleq \bigcup_{x \in \rho} \mathrm{supp}(\rho(x)) \end{aligned}$$

*Lemma 7.2* (*Support contains all relevant atoms*)

$$\forall v, \tau. \quad \begin{array}{r} \mathrm{fa}(\tau, v) \subseteq \mathrm{supp}(v) \text{ and } \mathrm{fb}(\tau, v) \subseteq \mathrm{supp}(v) \\ \text{and } \mathrm{fr}(\tau, v) \subseteq \mathrm{supp}(v) \text{ and } \mathrm{xa}(\tau, v) \subseteq \mathrm{supp}(v) \end{array}$$

As noted in Section 2.3, we represent a substitution as a set of ordered pairs.

*Lemma 7.3* (*Substitutions can sometimes commute*)
If $\mathrm{dom}(\sigma) \# \mathrm{dom}(\sigma')$ and $\mathrm{dom}(\sigma) \# \mathrm{rng}(\sigma')$ and $\mathrm{dom}(\sigma') \# \mathrm{rng}(\sigma)$, then $\sigma(\sigma'(v)) = \sigma'(\sigma(v))$.

#### 7.1.2 Semantics of β

Here, we list some properties of our $[\![\beta]\!]\,(\_)$ operation, especially the way that it relates to the substitutions or sets of atoms that it takes as input. All of these proofs are easy inductive arguments on $\beta$.

*Lemma 7.4* ($\beta$ *of* $\sigma$ *is consistent with its inputs*)
$\mathrm{dom}\left([\![\beta]\!]\,(\sigma_i)_i\right) = \bigcup_{i \,\hat{\in}\, \beta} \mathrm{dom}(\sigma_i)$ and $[\![\beta]\!]\,(\sigma_i)_i \subseteq \bigcup_{i \,\hat{\in}\, \beta} \sigma_i$

*Lemma 7.5* ($\beta$ *of* $A$ *is consistent with its inputs*)
$[\![\beta]\!]\,(A_i)_i = \bigcup_{i \,\hat{\in}\, \beta} A_i$

*Lemma 7.6* ($\beta$ *of* $\sigma$ *with disjoint domains does not shadow anything.*)
$\left(\forall i, j \,\hat{\in}\, \beta.\ i \neq j \Rightarrow \mathrm{dom}(\sigma_i) \# \mathrm{dom}(\sigma_j)\right)$ iff $[\![\beta]\!]\,(\sigma_i)_i = \bigcup_{i \,\hat{\in}\, \beta} \sigma_i$

*Lemma 7.7* ($\beta$ *of* $\sigma$ *ignores the* $\sigma$*s not mentioned by* $\beta$)
If $\forall i \,\hat{\in}\, \beta.\ \sigma_i = \sigma'_i$, then $[\![\beta]\!]\,(\sigma_i)_i = [\![\beta]\!]\,(\sigma'_i)_i$

*Lemma 7.8* ($\beta$ *of injective, mutually-range-disjoint* $\sigma$*s has an injective result*)
If $\forall i.\ \mathrm{inj}(\sigma_i)$, and $\forall i \neq j.\ \mathrm{rng}(\sigma_i) \# \mathrm{rng}(\sigma_j)$, then $\mathrm{inj}\left([\![\beta]\!]\,(\sigma_i)_i\right)$.

*Lemma 7.9* ($\beta$ *of a set respects renamings of that set*)
$[\![\beta]\!]\,\left(A_i\left[a'/a\right]\right)_i = [\![\beta]\!]\,(A_i)_i\left[a'/a\right]$

*Lemma 7.10* ($\beta$ *commutes with dom*)
$\mathrm{dom}\left([\![\beta]\!]\,(\sigma_i)_i\right) = [\![\beta]\!]\,(\mathrm{dom}(\sigma_i))_i$

https://doi.org/10.1017/S0956796816000137 Published online by Cambridge University Press

### *7.2 Basic α-equivalence lemmas*

Now, we show some basic properties related to α-equivalence and $\bowtie$. First, we need to define $=_{\text{shape}}$. Attempting to compare two values for α-equivalence is only interesting if they have the same shape.

$$v =_{\text{shape}} v' \subseteq \mathit{Value} \times \mathit{Value}$$

$$\frac{}{a =_{\text{shape}} a'} \qquad \frac{v =_{\text{shape}} v'}{\mathbf{inj0}(v) =_{\text{shape}} \mathbf{inj0}(v')} \qquad \frac{v =_{\text{shape}} v'}{\mathbf{inj1}(v) =_{\text{shape}} \mathbf{inj1}(v')}$$

$$\frac{\forall i.\ v_i =_{\text{shape}} v_i'}{\mathbf{prod}\,(v_i)_i =_{\text{shape}} \mathbf{prod}\,\left(v_i'\right)_i}$$

*Lemma 7.11* ($=_{\text{shape}}$ *is reflexive, preserved under renaming, and implied by* $=_\alpha$)
$\forall v.\ v =_{\text{shape}} v$ and $\forall v, a, a'.\ v =_{\text{shape}} v\left[a'/a\right]$ and $\forall \tau.\ \forall v =_\alpha v' : \tau.\ v =_{\text{shape}} v'$

The $\bowtie$ operator helps us relate exported binders in corresponding positions in different values. It is often applied to the respective subterms of two values whose α-equivalence is in question.

*Lemma 7.12* ($\bowtie$ *cannot fail, and outputs arbitrary* $\sigma$*s*)
If $v =_{\text{shape}} v'$ and $v : \tau$ and $A$ is finite, then $\exists \sigma, \sigma'.\ v \bowtie v' : \tau \to \sigma \bowtie \sigma'$ and $\text{inj}(\sigma)$ and $\text{inj}\left(\sigma'\right)$ and $\text{rng}(\sigma) \# A$ and $\text{rng}\left(\sigma'\right) \# A$

*Lemma 7.13* ($\bowtie$ *is about binders*)
If $v \bowtie v' : \tau \to \sigma \bowtie \sigma'$, then $\text{dom}(\sigma) = \text{fb}(\tau, v)$ and $\text{dom}\left(\sigma'\right) = \text{fb}\left(\tau, v'\right)$

Next, we show that α-equivalence is an equivalence. Showing transitivity is most complex because it depends on properties of substitutions, but even it is relatively straightforward.

*Lemma 7.14* (*Reflexivity of* α*-equivalence*)
$v =_{\text{R}} v : \tau$ and $v =_{\text{B}} v : \tau$ and $v =_\alpha v : \tau$

*Lemma 7.15* (*Symmetry of* α*-equivalence*)
$v =_\alpha v' : \tau$ implies $v' =_\alpha v : \tau$.

*Lemma 7.16* (*Transitivity of* α*-equivalence*)
$v =_\alpha v' : \tau$ and $v' =_\alpha v'' : \tau$ implies $v =_\alpha v'' : \tau$.

We need an operation on substitutions that can cancel out single-atom renamings on values (see Lemma 7.18). We call it "⊗":

$$\text{If } a' \notin \text{dom}(\sigma),\ \sigma \otimes \left[a'/a\right] \triangleq \left\{ \left\langle a_{\text{d-old}}\left[a'/a\right], a_{\text{r-old}} \right\rangle \;\middle|\; \left\langle a_{\text{d-old}}, a_{\text{r-old}} \right\rangle \in \sigma \right\}$$

*Lemma 7.17* ($\otimes$ *can be a no-op*)
If $a \notin \text{dom}(\sigma)$, then $\sigma \otimes \left[a'/a\right] = \sigma$.

https://doi.org/10.1017/S0956796816000137 Published online by Cambridge University Press

*Lemma 7.18* (⊗ *can undo substitutions*)
If $a \in \mathrm{dom}(\sigma)$ and $a' \# v$, then $\sigma \otimes \left[a'/a\right]\left(v\left[a'/a\right]\right) = \sigma(v)$.

*Lemma 7.19* (⊗ *adjusts domains*)
$a_{\mathrm{d}}\left[a'/a\right] \in \mathrm{dom}\left(\sigma \otimes \left[a'/a\right]\right)$ iff $a_{\mathrm{d}} \in \mathrm{dom}(\sigma)$

*Lemma 7.20* (⊗ *can distribute over* ▷)
If $a' \notin \mathrm{dom}(\sigma) \cup \mathrm{dom}\left(\sigma'\right)$, then $\left(\sigma \triangleright \sigma'\right) \otimes \left[a'/a\right] = \left(\sigma \otimes \left[a'/a\right]\right) \triangleright \left(\sigma' \otimes \left[a'/a\right]\right)$

*Lemma 7.21* (⊗ *commutes with* $\beta$)
$[\![\beta]\!]\left(\sigma_i \otimes \left[a'/a_i\right]\right)_i = [\![\beta]\!]\left(\sigma_i\right)_i \otimes \left[a'/a\right]$

*Lemma 7.22* (⋈ *reflects substitutions*)
If $v\left[a'/a\right] \bowtie v : \tau \to \sigma \bowtie \sigma'$ and $\mathrm{rng}(\sigma) \# v\left[a'/a\right], v$, then $\sigma = \sigma' \otimes \left[a'/a\right]$

*Lemma 7.23* (⋈ *reflects substitution, generalized*)
If $v \bowtie v' : \tau \to \sigma \bowtie \sigma'$, then $v\left[a'/a\right] \bowtie v' : \tau \to \sigma \otimes \left[a'/a\right] \bowtie \sigma'$

*Lemma 7.24* (*Substitution commutes with free binders*)
$\mathrm{fb}(\tau, \sigma(v)) = \sigma(\mathrm{fb}(\tau, v))$

*Lemma 7.25* (*Substitution commutes with free references*)
If $\mathrm{rng}(\sigma) \# v$ and $\mathrm{inj}(\sigma)$, then $\mathrm{fr}(\tau, \sigma(v)) = \sigma(\mathrm{fr}(\tau, v))$

*Lemma 7.26* (*Subsitution commutes with free atoms*)
If $\mathrm{rng}(\sigma) \# v$ and $\mathrm{inj}(\sigma)$, then $\mathrm{fa}(\tau, \sigma(v)) = \sigma(\mathrm{fa}(\tau, v))$

*Lemma 7.27* (*Substitution commutes with exposable atoms*)
If $\mathrm{rng}(\sigma) \# \mathbf{prod}\left(v_i\right)_i$ and $\mathrm{inj}(\sigma)$, then
$\mathrm{xa}\left(\mathrm{Prod}^{\Uparrow\beta_{\mathrm{ex}}}\left(\tau_i{\downarrow}v_i\right)_i, \sigma\left(\mathbf{prod}\left(v_i\right)_i\right)\right) = \sigma\left(\mathrm{xa}\left(\mathrm{Prod}^{\Uparrow\beta_{\mathrm{ex}}}\left(\tau_i{\downarrow}v_i\right)_i, \mathbf{prod}\left(v_i\right)_i\right)\right)$

Now, we need to relate free atoms and α-equivalence. This proof is more involved, so we set it out in more detail:

*Lemma 7.28* (*Renaming references respects reference-equivalence iff they are not free*)
Suppose $a' \# v$. Then, $a \notin \mathrm{fr}(\tau, v)$ if and only if $v\left[a'/a\right] =_{\mathrm{R}} v : \tau$.

*Proof*
By induction on the size of $v$ (all cases are trivial except for products and atoms):

1. Suppose $v = a_{\mathrm{v}}$ and $\tau = \mathrm{RAtom}$.

| | |
|---|---|
| $\mathrm{fr}(\mathrm{RAtom}, a_{\mathrm{v}}) = \{a_{\mathrm{v}}\}$ | By def. of fr. |
| $a \notin \mathrm{fr}(\mathrm{RAtom}, a_{\mathrm{v}})$ iff $v\left[a'/a\right] =_{\mathrm{R}} a_{\mathrm{v}} : \mathrm{RAtom}$ | By Rα-RAtom. |

2. Suppose $v = a_{\mathrm{v}}$ and $\tau = \mathrm{BAtom}$.

| | |
|---|---|
| $\mathrm{fr}(\mathrm{BAtom}, a_{\mathrm{v}}) = \varnothing$ | By def. of fr. |
| $a \notin \mathrm{fr}(\mathrm{BAtom}, a_{\mathrm{v}})$ iff $v\left[a'/a\right] =_{\mathrm{R}} a_{\mathrm{v}} : \mathrm{BAtom}$ | Trivially, by Rα-BAtom. |

https://doi.org/10.1017/S0956796816000137 Published online by Cambridge University Press

3. Suppose $v = \mathbf{prod}\,(v_i)_i$ and $\tau = \mathrm{Prod}^{\Uparrow\beta_{\mathrm{ex}}}\,(\tau_i{\downarrow}\beta_i)_i$.

First, we define (motivated by Rα-Prod):

$$P\left((\sigma_i)_i, \left(\sigma'_i\right)_i\right) \triangleq \forall i.\ \begin{array}{l} v_{\mathrm{sub}}\left[a'/a\right] \bowtie v_{\mathrm{sub},i} : \tau_{\mathrm{sub},i} \to \sigma_i \bowtie \sigma'_i \\ \quad \text{and } \mathrm{rng}(\sigma_i), \mathrm{rng}\left(\sigma'_i\right) \# v\left[a'/a\right], v \end{array}$$

First, we prove a sub-lemma that applies the induction hypothesis in the cases where $a$ is not imported into a particular subterm.

Assume for the sake of argument:

$$P\left((\sigma_i)_i, \left(\sigma'_i\right)_i\right) \tag{7.1}$$

Fix an $i$. We know,

$\sigma_i = \sigma'_i \otimes \left[a'/a\right]$ — By Lemma 7.22.

$[\![\beta_i]\!]\,(\sigma_j)_j = [\![\beta_i]\!]\,\left(\sigma'_j\right)_j \otimes \left[a'/a\right]$ — By Lemma 7.21.

(7.2)

And

$\mathrm{fb}\left(\tau_{\mathrm{sub},i}, v_{\mathrm{sub},i}\right) = \mathrm{dom}\left(\sigma'_i\right)$ — By Lemma 7.13.

Further, assume for the sake of argument:

$$a \notin [\![\beta_i]\!]\,\left(\mathrm{fb}\left(\tau_{\mathrm{sub},j}, v_{\mathrm{sub},j}\right)\right)_j \tag{7.3}$$

$a \notin [\![\beta_i]\!]\,\left(\mathrm{dom}\left(\sigma'_j\right)\right)_j$ — By Lemma 7.13.

$a \notin \mathrm{dom}\left([\![\beta_i]\!]\,\left(\sigma'_j\right)_j\right)$ — By Lemma 7.10.

$[\![\beta_i]\!]\,(\sigma_j)_j = [\![\beta_i]\!]\,\left(\sigma'_j\right)_j$ — By 7.2 and Lemma 7.17.

$\left([\![\beta_i]\!]\,(\sigma_j)_j\right)(v_i) = \left([\![\beta_i]\!]\,\left(\sigma'_j\right)_j\right)(v_i)$ — Trivially.

$\left([\![\beta_i]\!]\,(\sigma_j)_j(v_i)\right)\left[a'/a\right] =_{\mathrm{R}} [\![\beta_i]\!]\,\left(\sigma'_j\right)_j(v_i) : \tau_i$ iff $a \notin \mathrm{fr}(\tau_i, v_i)$ — By IH.

(7.4)

$a \# v$ or $a \notin \mathrm{rng}(\sigma_i)$ — By 7.1.

(7.5)

$a' \notin \mathrm{dom}(\sigma_i)$ — Because $a' \# v$.

$\left([\![\beta_i]\!]\,(\sigma_j)_j(v_i)\right)\left[a'/a\right] = [\![\beta_i]\!]\,(\sigma_j)_j\left(v_i\left[a'/a\right]\right)$ — By 7.5.

$[\![\beta_i]\!]\,(\sigma_j)_j\left(v_i\left[a'/a\right]\right) =_{\mathrm{R}} [\![\beta_i]\!]\,\left(\sigma'_j\right)_j(v_i) : \tau_i$ iff $a \notin \mathrm{fr}(\tau_i, v_i)$ — By 7.4.

https://doi.org/10.1017/S0956796816000137 Published online by Cambridge University Press

So

$$\forall i.\ \left(\begin{array}{l} \quad P\left(\left(\sigma_j\right)_j, \left(\sigma'_j\right)_j\right) \\ \text{and}\ \ a \notin [\![\beta_i]\!]\left(\mathrm{fb}\left(\tau_{\mathrm{sub},j}, v_{\mathrm{sub},j}\right)\right)_j \\ \Rightarrow [\![\beta_i]\!]\left(\sigma_j\right)_j\left(v_i\left[a'/a\right]\right) =_{\mathrm{R}} [\![\beta_i]\!]\left(\sigma'_j\right)_j(v_i) : \tau_i \\ \quad \text{iff}\ a \notin \mathrm{fr}(\tau_i, v_i) \end{array}\right) \quad \text{Unassuming 7.1 and 7.3.} \tag{7.6}$$

Now, we actually show the product case, which has another two layers of case analysis:

a. Suppose that: $a \in \mathrm{fr}(\tau, v)$

$\exists i.\ a \in \mathrm{fr}(\tau_i, v_i)$ and $a \notin [\![\beta_i]\!]\left(\mathrm{fb}\left(\tau_{\mathrm{sub},j}, v_{\mathrm{sub},j}\right)\right)_j$ By def. of fr. (7.7)

$P\left(\left(\sigma_j\right)_j, \left(\sigma'_j\right)_j\right)$ By Lemma 7.12.

But

$\exists i.\ [\![\beta_i]\!]\left(\sigma_j\right)_j\left(v_i\left[a'/a\right]\right) \neq_{\mathrm{R}} [\![\beta_i]\!]\left(\sigma'_j\right)_j(v_i) : \tau_i$ By 7.6 and 7.7.

So

$a \in \mathrm{fr}(\tau, v) \Rightarrow v\left[a'/a\right] \neq_{\mathrm{R}} v : \tau$ By Rα-Prod and supposition. (7.8)

b. Suppose that: $a \notin \mathrm{fr}(\tau, v)$

$\forall i.\ a \notin \mathrm{fr}(\tau_i, v_i)$ or $a \in [\![\beta_i]\!]\left(\mathrm{fb}\left(\tau_{\mathrm{sub},j}, v_{\mathrm{sub},j}\right)\right)_j$ By def. of fr. (7.9)

We can have $\forall j.\ \exists \sigma_j, \sigma'_j.$ :

$P\left(\left(\sigma_j\right)_j, \left(\sigma'_j\right)_j\right)$ By Lemmas 7.12 and 7.11. (7.10)

Fix an $i$.

i. Suppose that $a \notin [\![\beta_i]\!]\left(\mathrm{fb}\left(\tau_{\mathrm{sub},j}, v_{\mathrm{sub},j}\right)\right)_j$:

$a \notin \mathrm{fr}(\tau_i, v_i)$ By definition of fr. (7.11)

$[\![\beta_i]\!]\left(\sigma_j\right)_j\left(v_i\left[a'/a\right]\right) =_{\mathrm{R}} [\![\beta_i]\!]\left(\sigma'_j\right)_j(v_i) : \tau_i$ By 7.6 and 7.10. (7.12)

ii. Suppose that $a \in [\![\beta_i]\!]\left(\mathrm{fb}\left(\tau_{\mathrm{sub},j}, v_{\mathrm{sub},j}\right)\right)_j$.
In this case, we do not need the IH, because the substitutions generated by $\bowtie$ already account for the renaming:

$[\![\beta_i]\!]\left(\sigma_j\right)_j = [\![\beta_i]\!]\left(\sigma'_j\right)_j \otimes \left[a'/a\right]$ By 7.10 and Lemmas 7.22 and 7.21.

$[\![\beta_i]\!]\left(\sigma_j\right)_j\left(v_i\left[a'/a\right]\right) = [\![\beta_i]\!]\left(\sigma'_j\right)_j(v_i)$ By Lemma 7.18.

$[\![\beta_i]\!]\left(\sigma_j\right)_j\left(v_i\left[a'/a\right]\right) =_{\mathrm{R}} [\![\beta_i]\!]\left(\sigma'_j\right)_j(v_i) : \tau_i$ By Lemma 7.14.

So

$a \notin \mathrm{fr}(\tau, v) \Rightarrow v\left[a'/a\right] =_{\mathrm{R}} v : \tau$ By 7.12.

https://doi.org/10.1017/S0956796816000137 Published online by Cambridge University Press

And so

$a \in \mathrm{fr}(\tau, v)$ iff $v\,[a_\mathrm{v}/a] =_\alpha a_\mathrm{v} : \tau$ By 7.8.

□

*Lemma 7.29* (*Renaming binders respects binder equivalence iff they are not free*)
Suppose $a' \# v$. Then, $a \notin \mathrm{fb}(\tau, v)$ iff $v\,[a'/a] =_\mathrm{B} v : \tau$.

*Proof*
By induction on $v$ (all cases are trivial except for products and atoms):

1. Suppose $v = a_\mathrm{v}$ and $\tau = \mathrm{BAtom}$:

   $\mathrm{fb}(\mathrm{BAtom}, v) = \{a_\mathrm{v}\}$ By def. of fb.

   $a \in \{a_\mathrm{v}\}$ iff $v\,[a'/a] =_\mathrm{B} v : \tau$ By Bα-BAtom.

2. Suppose $v = a_\mathrm{v}$ and $\tau = \mathrm{RAtom}$:

   $\mathrm{fb}(\mathrm{BAtom}, a_\mathrm{v}) = \varnothing$ By def. of fb.

   $a \notin \mathrm{fb}(\mathrm{BAtom}, a_\mathrm{v})$ iff $v\,[a'/a] =_\mathrm{B} a_\mathrm{v} : \mathrm{BAtom}$ Trivially, by Bα-BAtom.

3. Suppose $v = \mathbf{prod}\,(v_i)_i$ and $\tau = \mathrm{Prod}^{\Uparrow\beta_\mathrm{ex}}\,(\tau_i{\downarrow}\beta_i)_i$:

   $a \in \mathrm{fb}(\tau, v)$ iff $a \in [\![\beta_\mathrm{ex}]\!]\,(\mathrm{fb}(\tau_i, v_i))_i$ By def. of fb.

   iff $\exists i \mathrel{\hat{\in}} \beta_\mathrm{ex}.\ a \in \mathrm{fb}(\tau_i, v_i)$ By a straightforward induction on $\beta$.

   iff $\exists i \mathrel{\hat{\in}} \beta_\mathrm{ex}.\ v_i\,[a'/a] \neq_\mathrm{B} v_i : \tau_i$ By IH.

   iff $v\,[a'/a] \neq_\mathrm{B} v : \tau$ By Bα-Prod.

□

*Lemma 7.30* (*Renaming atoms respect α-equivalence iff they are not free*)
Suppose $a' \# v$. Then, $a \notin \mathrm{fa}(\tau, v)$ if and only if $v\,[a'/a] =_\alpha v : \tau$.

*Proof*
Follows from Lemmas 7.28 and 7.29 and αEq. □

Now, we prove some corollaries of those lemmas.

*Lemma 7.31* (*Non-free atoms can be mass-renamed*)
Suppose $\mathrm{inj}(\sigma)$ and $\mathrm{rng}(\sigma) \# v$ and $\mathrm{dom}(\sigma) \# \mathrm{rng}(\sigma)$.
$\mathrm{dom}(\sigma) \# \mathrm{fb}(\tau, v) \Rightarrow \sigma(v) =_\mathrm{B} v : \tau$
$\mathrm{dom}(\sigma) \# \mathrm{fr}(\tau, v) \Rightarrow \sigma(v) =_\mathrm{R} v : \tau$
$\mathrm{dom}(\sigma) \# \mathrm{fa}(\tau, v) \Rightarrow \sigma(v) =_\alpha v : \tau$

*Lemma 7.32* (*Good renamings preserve binder equivalence*)
Suppose $a' \# v, v'$. Then, $v =_\mathrm{B} v' : \tau \Rightarrow v\,[a'/a] =_\mathrm{B} v'\,[a'/a] : \tau$

*Lemma 7.33* (*Good renamings preserve reference equivalence*)
Suppose $a' \# v, v'$. Then, $v =_\mathrm{R} v' : \tau \Rightarrow v\,[a'/a] =_\mathrm{R} v'\,[a'/a] : \tau$

https://doi.org/10.1017/S0956796816000137 Published online by Cambridge University Press

*Lemma 7.34* (*Good renamings preserve α-equivalence*)
Suppose $a' \# v, v'$. Then, $v =_\alpha v' : \tau \Rightarrow v\left[a'/a\right] =_\alpha v'\left[a'/a\right] : \tau$

*Lemma 7.35* (*Good substitutions preserve α-equivalence*)
Suppose $\mathrm{rng}(\sigma) \# v, v'$ and $\mathrm{dom}(\sigma) \# \mathrm{rng}(\sigma)$ and $\mathrm{inj}(\sigma)$.
$v =_\mathrm{B} v' : \tau \Rightarrow \sigma(v) =_\mathrm{B} \sigma\left(v'\right) : \tau$
$v =_\mathrm{R} v' : \tau \Rightarrow \sigma(v) =_\mathrm{R} \sigma\left(v'\right) : \tau$
$v =_\alpha v' : \tau \Rightarrow \sigma(v) =_\alpha \sigma\left(v'\right) : \tau$

*Lemma 7.36* (*Free atoms are the same for α-equivalent values*)
$v =_\mathrm{B} v' : \tau \Rightarrow \mathrm{fb}(\tau, v) = \mathrm{fb}\left(\tau, v'\right)$
$v =_\mathrm{R} v' : \tau \Rightarrow \mathrm{fr}(\tau, v) = \mathrm{fr}\left(\tau, v'\right)$
$v =_\alpha v' : \tau \Rightarrow \mathrm{fa}(\tau, v) = \mathrm{fa}\left(\tau, v'\right)$

The following are helper lemmas for Lemma 7.42.

*Lemma 7.37* (*Binder-equivalent values don't need their free binders renamed*)
Suppose $v \bowtie v' : \tau \to \sigma \bowtie \sigma'$. If $v =_\mathrm{B} v' : \tau$, then $\sigma = \sigma'$.

*Lemma 7.38* (*Identical renamings can be removed, retaining reference equality*)
If $\sigma(v) =_\mathrm{R} \sigma'\left(v'\right) : \tau$ and $\sigma(a) = a' = \sigma'(a)$ and $a' \# v, v'$,
then $\left(\sigma \setminus \{a\}\right)(v) =_\mathrm{R} \left(\sigma' \setminus \{a\}\right)\left(v'\right) : \tau$

*Lemma 7.39* (*Identical renamings can be removed, retaining binder equality*)
If $\sigma(v) =_\mathrm{B} \sigma'\left(v'\right) : \tau$ and $\sigma(a) = a' = \sigma'(a)$ and $a' \# v, v'$,
then $\left(\sigma \setminus \{a\}\right)(v) =_\mathrm{B} \left(\sigma' \setminus \{a\}\right)\left(v'\right) : \tau$

*Lemma 7.40* (*Disjointness of unrelated binders*)
Let $v = \mathbf{prod}\,(v_i)_i$ and $\tau = \mathrm{Prod}^{\Uparrow\beta_\mathrm{ex}}\,(\tau_i{\downarrow}\beta_i)_i$ and $\sigma_\mathrm{all} = \bigcup_i \sigma'_i$.

Suppose the following: $\forall i.\ \sigma'_i \subseteq \sigma_i$ and $\mathrm{dom}(\sigma_\mathrm{all}) \# \mathrm{fr}(\tau, v)$ and $\forall i \neq j.\ \mathrm{dom}(\sigma_i) \# \mathrm{dom}\left(\sigma_j\right)$ and $\forall i.\ \mathrm{fb}(\tau_i, v_i) = \mathrm{dom}(\sigma_i)$.

Then, $\forall i.\ \mathrm{dom}(\sigma_\mathrm{all}) \setminus \mathrm{dom}\left(\left[\!\left[\beta_i\right]\!\right]\left(\sigma'_j\right)_j\right) \# \mathrm{fr}(\tau_i, v_i)$.

*Lemma 7.41* (*$\bowtie$ makes pairs of binder-disjoint values binder-equal*)
If $\vdash_\text{bndrs-disj} v : \tau$ and $\vdash_\text{bndrs-disj} v' : \tau$, then

$$v \bowtie v' : \tau \to \sigma \bowtie \sigma' \Rightarrow \sigma(v) =_\mathrm{B} \sigma'\left(v'\right) : \tau.$$

The purpose of the following lemma is to show that any pair of α-equivalent, sufficiently disjoint values are suitable for unpacking into a pair of environments that contain the free names $A$. By this, we mean that a single pair of substitutions can make their *subterms* α-equivalent, not one pair per pair of subterms.

*Lemma 7.42* (*Sufficiently disjoint α-equivalent values can be opened*)
Let $\tau_\mathrm{obj} = \mathrm{Prod}^{\Uparrow\beta_\mathrm{ex}}\,(\tau_i{\downarrow}\beta_i)_i$ and $v = \mathbf{prod}\,(v_i)_i$ and $v' = \mathbf{prod}\,\left(v'_i\right)_i$.

Suppose $\mathrm{fa}\left(\tau_\mathrm{obj}, v\right) \vdash_\text{suff-disj} v : \tau_\mathrm{obj}$ and $\mathrm{fa}\left(\tau_\mathrm{obj}, v'\right) \vdash_\text{suff-disj} v' : \tau_\mathrm{obj}$ and $A$ is finite.

If $v =_\alpha v' : \tau_\mathrm{obj}$, then $\exists \sigma, \sigma'.\ \mathrm{dom}(\sigma) \subseteq \mathrm{xa}\left(\tau_\mathrm{obj}, v\right)$ and $\mathrm{dom}\left(\sigma'\right) \subseteq \mathrm{xa}\left(\tau_\mathrm{obj}, v'\right)$ and $\forall i.\ \sigma(v_i) =_\alpha \sigma'\left(v'_i\right) : \tau_i$ and $\mathrm{rng}(\sigma) \# A$ and $\mathrm{rng}\left(\sigma'\right) \# A$ and $\mathrm{inj}(\sigma)$ and $\mathrm{inj}\left(\sigma'\right)$

https://doi.org/10.1017/S0956796816000137 Published online by Cambridge University Press

*Proof*

$\forall i.\ v_i \bowtie v_i' : \tau_i \to \sigma_{\text{orig},i} \bowtie \sigma'_{\text{orig},i}$ By Rα-Prod. (7.13)

$\forall i, j.\ \text{rng}(\sigma_{\text{orig},i}), \text{rng}(\sigma'_{\text{orig},i}) \# v_j, v_j'$ Likewise. (7.14)

$\forall i.\ [\![\beta_i]\!]\ (\sigma_{\text{orig},j})_j(v_i) =_{\text{R}} [\![\beta_i]\!]\ (\sigma'_{\text{orig},j})_j(v_i') : \tau_i$ Likewise. (7.15)

To avoid collisions, choose $\sigma_{\text{disj}}$:

$$\text{dom}(\sigma_{\text{disj}}) = \bigcup_i \text{rng}(\sigma_{\text{orig},i}) \cup \text{rng}\left(\sigma'_{\text{orig},i}\right)$$
$$\text{inj}(\sigma_{\text{disj}}) \text{ and } \forall i.\ \text{rng}(\sigma_{\text{disj}}) \# v_i, v_i' \text{ and } \text{rng}(\sigma_{\text{disj}}) \# A \quad (7.16)$$

To keep exported binders fixed, define

$\forall i.\ \sigma_i \triangleq \sigma_{\text{disj}} \circ \sigma_{\text{orig},i}$ and $\forall i.\ \sigma_i' \triangleq \sigma_{\text{disj}} \circ \sigma'_{\text{orig},i}$ (7.17)

Define

$$\sigma_{\text{local},i} \triangleq \begin{cases} \varnothing & i \hat{\in} \beta_{\text{ex}} \\ \sigma_i & i \hat{\notin} \beta_{\text{ex}} \end{cases} \text{ and } \sigma'_{\text{local},i} \triangleq \begin{cases} \varnothing & i \hat{\in} \beta_{\text{ex}} \\ \sigma_i' & i \hat{\notin} \beta_{\text{ex}} \end{cases} \quad (7.18)$$

(7.19)

Now,

$$\forall i, j.\quad \text{rng}(\sigma_{\text{local},i}), \text{rng}\left(\sigma'_{\text{local},i}\right) \# v_j, v_j', A \quad \text{and} \quad \text{inj}(\sigma_{\text{local},i}) \quad \text{By 7.14 and 7.16.} \quad (7.20)$$

(7.21)

For convenience, define

$\sigma_{\text{imp},i} \triangleq [\![\beta_i]\!]\ (\sigma_{\text{local},j})_j$ and $\sigma'_{\text{imp},i} \triangleq [\![\beta_i]\!]\ (\sigma'_{\text{local},j})_j$ (7.22)

$\forall i \hat{\notin} \beta_{\text{ex}}.\ \sigma_{\text{imp},i}(v_i) =_{\text{R}} \sigma'_{\text{imp},i}(v_i') : \tau_i$ By 7.15. (7.23)

$\text{dom}(\sigma_{\text{local},i}) \subseteq \text{fb}(\tau_i, v_i)$ and $\text{dom}(\sigma'_{\text{local},i}) \subseteq \text{fb}(\tau_i, v_i')$ By 7.13 and Lemma 7.13. (7.24)

But

$v =_{\text{B}} v' : \tau_{\text{obj}}$ Because $v =_\alpha v' : \tau_{\text{obj}}$.

$\forall i \hat{\in} \beta_{\text{ex}}.\ v_i =_{\text{B}} v_i' : \tau_i$ By def. of $=_{\text{B}}$.

$\forall i \hat{\in} \beta_{\text{ex}}.\ \sigma_i = \sigma_i'$ By Lemma 7.37. (7.25)

So it doesn't matter whether a subterm is exported:

$\forall i.\ \sigma_{\text{imp},i}(v_i) =_{\text{R}} \sigma'_{\text{imp},i}(v_i') : \tau_i$ By 7.23, 7.18, and 7.20 and using Lemma 7.38 once for each $a \in \text{dom}(\sigma_i)$, where $i \hat{\in} \beta_{\text{ex}}$. (7.26)

Now,

$\forall i, j \in \beta_{\text{ex}}.\ i \neq j \Rightarrow \text{dom}(\sigma_{\text{local},i}) \# \text{dom}(\sigma_{\text{local},j})$ By def. of $\vdash_{\text{suff-disj}}$ and 7.24. (7.27)

And

$\forall i \neq j.\ \text{dom}(\sigma_{\text{local},i}) \# \text{dom}(\sigma_{\text{local},j})$ By 7.18. (7.28)

https://doi.org/10.1017/S0956796816000137 Published online by Cambridge University Press

So, define a single substitution:

$$\sigma \triangleq \biguplus_i \sigma_{\text{local},i} \tag{7.29}$$

$\text{dom}(\sigma) \# \text{rng}(\sigma)$ By 7.20. (7.30)

But:

$\text{dom}(\sigma) \# \text{fb}(\tau_{\text{obj}}, v)$ By def. of fb. (7.31)

$\text{dom}(\sigma) \subseteq \text{xa}(\tau_{\text{obj}}, v)$ By def. of xa. (7.32)

$\text{dom}(\sigma) \# \text{fa}(\tau_{\text{obj}}, v)$ By def. of $\vdash_{\text{suff-disj}}$.

$\text{dom}(\sigma) \# \text{fr}(\tau_{\text{obj}}, v)$ By def. of fa.

$\forall i.\ \text{dom}(\sigma) \setminus \text{dom}(\sigma_{\text{imp},i}) \# \text{fr}(\tau_i, v_i)$ By 7.28 on Lemma 7.40. (7.33)

And,

$\forall i.\ \text{dom}(\sigma) \setminus \text{dom}(\sigma_{\text{imp},i}) \# \bigcup_i \text{rng}(\sigma_i)$ By 7.20 and 7.24 and Lemma 7.2. (7.34)

But

$\forall i.\ \text{fr}(\tau_i, \sigma_{\text{imp},i}(v_i)) = \sigma_{\text{imp},i}(\text{fr}(\tau_i, v_i))$ By 7.20 and Lemma 7.25.

$\quad \subseteq (\text{fr}(\tau_i, v_i) \setminus \text{dom}(\sigma_{\text{imp},i})) \cup \text{rng}(\sigma_{\text{imp},i})$ By set arithmetic.

$\quad \subseteq \text{fr}(\tau_i, v_i) \cup \text{rng}(\sigma_{\text{imp},i})$ By set arithmetic.

$\quad \subseteq \text{fr}(\tau_i, v_i) \cup \bigcup_j \text{rng}(\sigma_j)$ By Lemma 7.5.

$\quad \# \text{dom}(\sigma) \setminus \text{dom}(\sigma_{\text{imp},i})$ By 7.33 and 7.34.

$\text{fr}(\tau_i, \sigma_{\text{imp},i}(v_i)) \# \text{dom}(\sigma \setminus \sigma_{\text{imp},i})$ By set arithmetic. (7.35)

(7.36)

$\sigma_{\text{imp},i} \subseteq \sigma$ By Lemma 7.4.

$\forall i.\ \sigma_{\text{imp},i}(v_i) =_{\text{R}} (\sigma \setminus \sigma_{\text{imp},i})(\sigma_{\text{imp},i}(v_i)) : \tau_i$ By 7.35 and 7.30 on Lemma 7.31.

$\forall i.\ \quad = \sigma(v_i)$ By 7.30. (7.37)

(7.38)

And,

$\sigma' \triangleq \biguplus_i \sigma'_{\text{local},i}$ and $\forall i.\ \sigma'_{\text{imp},i}(v'_i) =_{\text{R}} \sigma'(v'_i) : \tau_i$ By a similar argument.

$\forall i.\ \sigma(v_i) =_{\text{R}} \sigma'(v'_i) : \tau_i$ By 7.37 and 7.26 and Lemmas 7.28 and 7.16. (7.39)

Also,

$\forall i \hat{\in} \beta_{\text{ex}}.\ \text{dom}(\sigma) \# \text{fb}(\tau_i, v_i)$ and $\text{dom}(\sigma') \# \text{fb}(\tau_i, v'_i)$ By 7.31 and def. of fb and Lemma 7.5. (7.40)

$\forall i \hat{\in} \beta_{\text{ex}}.\ \sigma(v_i) =_{\text{B}} \sigma'(v'_i) : \tau_i$ By 7.20 and 7.30 on Lemma 7.31. (7.41)

https://doi.org/10.1017/S0956796816000137 Published online by Cambridge University Press

And,

$\forall i \hat{\notin} \beta_{\text{ex}}.\ \sigma_{\text{orig},i}(v_i) =_{\text{B}} \sigma'_{\text{orig},i}(v'_i) : \tau_i$ — By def. of $\vdash_{\text{suff-disj}}$ and 7.13 on Lemma 7.41. (7.42)

$\forall i \hat{\notin} \beta_{\text{ex}}.\ \sigma_i(v_i) =_{\text{B}} \sigma'_i(v'_i) : \tau_i$ — By 7.16 on Lemma 7.31.

$\forall i \hat{\notin} \beta_{\text{ex}}.\ \sigma_{\text{local},i}(v_i) =_{\text{B}} \sigma'_{\text{local},i}(v'_i) : \tau_i$ — Because $\forall i \hat{\notin} \beta_{\text{ex}}.\sigma_{\text{local},i} = \sigma_i$ and $\sigma'_{\text{local},i} = \sigma'_i$.

$\forall i \hat{\notin} \beta_{\text{ex}}.\ \sigma(v_i) =_{\text{B}} \sigma'(v'_i) : \tau_i$ — By 7.16 on Lemma 7.31. (7.43)

Finally,

$\forall i.\ \sigma(v_i) =_{\alpha} \sigma'(v'_i) : \tau_i$ — By 7.39 and 7.41.

Furthermore, we have:

$\text{dom}(\sigma) \subseteq \text{xa}(\tau_{\text{obj}}, v)$ and $\text{dom}(\sigma') \subseteq \text{xa}(\tau_{\text{obj}}, v')$ — By 7.32.

$\text{rng}(\sigma) \# A$ and $\text{rng}(\sigma') \# A$ — By 7.16.

$\text{inj}(\sigma)$ and $\text{inj}(\sigma')$ — By 7.13 and 7.16.

□

*Lemma 7.43* (*New, disjoint, binders can be generated*)
Given $\mathbf{prod}\,(v_i)_i : \text{Prod}\,(\tau_i{\downarrow}\beta_i)_i$, and finite $A$: $\forall$ i. $\exists\ \sigma_i$. such that $v_i \bowtie v_i : \tau_i \to \sigma_i \bowtie \sigma_i$ and $\text{inj}(\sigma_i)$ and $\text{rng}(\sigma_i) \# A$ and $\forall j \neq i.\ \text{rng}(\sigma_i) \# \text{rng}(\sigma_j)$ and $\forall j.\ \text{rng}(\sigma_i) \# \text{dom}(\sigma_j)$.

*Lemma 7.44* (*Products with binder-equivalent components are binder-equivalent*)
If $\forall i.\ v_i =_{\text{B}} v'_i : \tau_i$, then $\mathbf{prod}\,(v_i)_i =_{\text{B}} \mathbf{prod}\,(v'_i)_i : \text{Prod}^{\Uparrow\beta_{\text{ex}}}\,(\tau_i{\downarrow}\beta_i)_i$.

*Lemma 7.45* (*Products with α-equivalent components are α-equivalent*)
If $\forall i.\ v_i =_{\alpha} v'_i : \tau_i$, then $\mathbf{prod}\,(v_i)_i =_{\alpha} \mathbf{prod}\,(v'_i)_i : \text{Prod}^{\Uparrow\beta_{\text{ex}}}\,(\tau_i{\downarrow}\beta_i)_i$.

*Lemma 7.46* (*Substitutions don't affect* FAULT*'s uniqueness*)
$\sigma(w) = \text{FAULT}$ iff $w = \text{FAULT}$

*Lemma 7.47* (*Good substitutions are undoable*)
If $\text{inj}(\sigma)$ and $\text{rng}(\sigma) \# v$, then $v = \text{inv}(\sigma)(\sigma(v))$ and $\text{inj}(\text{inv}(\sigma))$ and $\text{rng}(\text{inv}(\sigma)) \# \sigma(v)$.

*Lemma 7.48* (*Good substitutions reflect* $\vdash_{\text{suff-disj}}$)
If $\text{inj}(\sigma)$ and $\text{rng}(\sigma) \# v$ and $\sigma(A) \vdash_{\text{suff-disj}} \sigma(v) : \tau$, then $A \vdash_{\text{suff-disj}} v : \tau$

*Lemma 7.49* (*Good substitutions preserve* $\vdash_{\text{suff-disj}}$)
If $\text{inj}(\sigma)$ and $\text{rng}(\sigma) \# A, v$ and $A \vdash_{\text{suff-disj}} v : \tau$, then $\sigma(A) \vdash_{\text{suff-disj}} \sigma(v) : \tau$.

*Lemma 7.50* (*Range update can be expressed in terms of composition*)
If $\text{rng}(\sigma) \# v$, then for all $\sigma_{\text{avoid}}$, we have $\{\langle a, \sigma_{\text{avoid}}(a')\rangle \mid \langle a, a'\rangle \in \sigma\}(v) = \sigma_{\text{avoid}}(\sigma(v))$

*Lemma 7.51* (*Simultaneous range updates respect α-equivalence*)
If $\text{rng}(\sigma) \# v$ and $\text{rng}(\sigma') \# v'$ and $\text{inj}(\sigma_{\text{avoid}})$ and $\text{rng}(\sigma_{\text{avoid}}) \# \sigma(v), \sigma'(v')$ and $\text{dom}(\sigma_{\text{avoid}}) \# \text{rng}(\sigma_{\text{avoid}})$, then

$$\left\{\langle a, \sigma_{\text{avoid}}(a')\rangle \;\middle|\; \langle a, a'\rangle \in \sigma\right\}(v) =_{\alpha} \left\{\langle a, \sigma_{\text{avoid}}(a')\rangle \;\middle|\; \langle a, a'\rangle \in \sigma'\right\}(v')$$
$$: \tau \text{ iff } \sigma(v) =_{\alpha} \sigma'(v') : \tau$$

https://doi.org/10.1017/S0956796816000137 Published online by Cambridge University Press

### 7.3 Execution soundness itself

*Theorem 7.1* (*Determinism up to α-equivalence, termination-insensitive version*)

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

*Proof*
By induction on $k$, and case analysis on $e$ in

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

- *Case* E-Fresh-⋆:

  First,

  $\Gamma \vdash_{\text{exe}} \langle (\textbf{fresh } x \textbf{ in } e), \rho \rangle \overset{k+1}{\Longrightarrow} w_{\text{out}}$
  $\Gamma \vdash_{\text{exe}} \langle (\textbf{fresh } x \textbf{ in } e), \rho' \rangle \overset{k'+1}{\Longrightarrow} w'_{\text{out}}$ — By hypothesis. (7.44)

  Define

  $\Gamma_{\text{IH}} \triangleq \Gamma, x{:}\text{BAtom}$

  We know,

  $\exists a.\ \Gamma_{\text{IH}} \vdash_{\text{exe}} \langle e, \rho[x \to a] \rangle \overset{k}{\Rightarrow} w$ — By inv. of E-Fresh-⋆. (7.45)

  $\exists a'.\ \Gamma_{\text{IH}} \vdash_{\text{exe}} \langle e, \rho'[x \to a'] \rangle \overset{k'}{\Rightarrow} w'$ — By inv. of E-Fresh-⋆. (7.46)

  $\text{inj}(\sigma)$ and $\text{inj}(\sigma')$
  $\text{rng}(\sigma) \# \rho, w, a$
  $\text{rng}(\sigma') \# \rho', w', a'$ — By hypothesis and inv. of E-Fresh-⋆. (7.47)

  $\sigma(\rho) =_\alpha \sigma'(\rho') : \Gamma$ — By hypothesis. (7.48)

  Choose

  $a_{\text{fr}} \# \sigma, \sigma', w, w', \rho, \rho', a, a'$ (7.49)

  Define

  $\sigma_{\text{IH}} \triangleq \{\langle \sigma(a), a_{\text{fr}} \rangle\} \circ \sigma$
  $\sigma'_{\text{IH}} \triangleq \{\langle \sigma'(a'), a_{\text{fr}} \rangle\} \circ \sigma'$ (7.50)

  $\sigma_{\text{IH}}(a) = \sigma'_{\text{IH}}(a')$ — By straightforward substitution. (7.51)

https://doi.org/10.1017/S0956796816000137 Published online by Cambridge University Press

And,

| | | |
|---|---|---|
| $a \notin \mathrm{fa}_{\mathrm{env}}(\Gamma, \rho)$ and $a' \notin \mathrm{fa}_{\mathrm{env}}(\Gamma, \rho')$ | By inv. of E-Fresh-$\star$. | |
| $\sigma(a) \notin \mathrm{fa}_{\mathrm{env}}(\Gamma, \sigma(\rho))$<br>$\sigma'(a') \notin \mathrm{fa}_{\mathrm{env}}(\Gamma, \sigma(\rho'))$ | By 7.47 and Lemma 7.26. | |
| $\sigma_{\mathrm{IH}}(\rho) =_\alpha \sigma'_{\mathrm{IH}}(\rho') : \Gamma$ | By 7.49 on Lemma 7.30. | |
| $\sigma_{\mathrm{IH}}(\rho\,[x \to a]) =_\alpha \sigma'_{\mathrm{IH}}(\rho'\,[x \to a']) : \Gamma_{\mathrm{IH}}$ | By 7.51. | (7.52) |

And,

| | | |
|---|---|---|
| $\mathrm{inj}(\sigma_{\mathrm{IH}})$ and $\mathrm{inj}(\sigma'_{\mathrm{IH}})$ | By 7.47 and 7.49. | (7.53) |
| $\mathrm{rng}(\sigma_{\mathrm{IH}}) \# \rho\,[x \to a], w$<br>$\mathrm{rng}(\sigma'_{\mathrm{IH}}) \# \rho'\,[x \to a'], w'$ | By 7.47 and 7.49. | (7.54) |

So

| | | |
|---|---|---|
| $\sigma_{\mathrm{IH}}(w) =_\alpha \sigma'_{\mathrm{IH}}(w') : \tau$ | By 7.52 and 7.53 on IH. | (7.55) |

Now, we need to show that the results are $\alpha$-equivalent under the original substitution.

1. Suppose that $w = \textsc{fault}$ or $w' = \textsc{fault}$:

| | | |
|---|---|---|
| $w = w' = \textsc{fault}$ | By 7.55 on Lemma 7.46. | (7.56) |
| $w_{\mathrm{out}} = \textsc{fault}$ and $w'_{\mathrm{out}} = \textsc{fault}$ | By 7.45 and 7.46 and E-Fresh-$\star$. | |

2. Suppose that $w \neq \textsc{fault}$ and $w' \neq \textsc{fault}$:

| | | |
|---|---|---|
| $\mathrm{fa}(\tau, \sigma_{\mathrm{IH}}(w)) = \mathrm{fa}(\tau, \sigma'_{\mathrm{IH}}(w'))$ | By 7.55 on Lemma 7.36. | (7.57) |
| $\sigma_{\mathrm{IH}}(\mathrm{fa}(\tau, w)) = \sigma'_{\mathrm{IH}}(\mathrm{fa}(\tau, w'))$ | By 7.53 and 7.54 on Lemma 7.26. | |
| $\sigma_{\mathrm{IH}}(a) \notin \mathrm{fa}(\tau, \sigma_{\mathrm{IH}}(w))$<br>iff $\sigma'_{\mathrm{IH}}(a') \notin \mathrm{fa}(\tau, \sigma'_{\mathrm{IH}}(w'))$ | By 7.51 and 7.53. | |
| $a \notin \mathrm{fa}(\tau, w)$ iff $a' \notin \mathrm{fa}(\tau, w')$ | By 7.53. | (7.58) |

   a. So, suppose that both names are not free:

| | | |
|---|---|---|
| $a \notin \mathrm{fa}(\tau, w)$ and $a' \notin \mathrm{fa}(\tau, w')$ | | (7.59) |
| $w_{\mathrm{out}} = w$ and $w'_{\mathrm{out}} = w'$ | By 7.45 and 7.46 and E-Fresh-$\star$. | (7.60) |

   And also,

| | | |
|---|---|---|
| $\sigma_\#(a) \notin \mathrm{fa}(\tau, \sigma_\#(w))$<br>and $\sigma'_\#(a') \notin \mathrm{fa}(\tau, \sigma'_\#(w'))$ | By 7.59 and 7.47 and Lemma 7.26. | (7.61) |
| $\sigma_\#(w) =_\alpha \sigma'_\#(w') : \tau$ | By 7.55 and 7.49 on Lemma 7.30. | |

https://doi.org/10.1017/S0956796816000137 Published online by Cambridge University Press

$\sigma(w) =_\alpha \sigma'(w') : \tau$ — By 7.49 on Lemma 7.51.

$\sigma(w_{\text{out}}) =_\alpha \sigma'(w'_{\text{out}}) : \tau$ — By 7.60. (7.62)

b. Alternatively, suppose $a \in \text{fa}(\tau, w)$ and $a' \in \text{fa}(\tau, w')$:

$w_{\text{out}} = \text{FAULT}$ and $w'_{\text{out}} = \text{FAULT}$ — By 7.45 and 7.46 and E-FRESH-$\star$.

- *Case* E-OPEN-$\star$: Let $v_{\text{obj}} \triangleq \textbf{prod}\,(v_i)_i$ and $v'_{\text{obj}} \triangleq \textbf{prod}\,(v'_i)_i$ and $e_{\text{all}} \triangleq (\textbf{open}\ x_{\text{obj}}\ ((x_i)_i)\ e)$.

First,

$\Gamma \vdash_{\text{exe}} \langle e_{\text{all}}, \rho\rangle \overset{k+1}{\Longrightarrow} w_{\text{out}}$ and $\Gamma \vdash_{\text{exe}} \langle e_{\text{all}}, \rho'\rangle \overset{k'+1}{\Longrightarrow} w'_{\text{out}}$ — By hypothesis. (7.63)

Define

$\Gamma_{\text{IH}} \triangleq \Gamma, (x_i{:}\tau_i)_i$

Now,

$\forall i.\ \exists v_i.\ \Gamma_{\text{IH}} \vdash_{\text{exe}} \langle e, \rho\,[x_i \to v_i]_i\rangle \overset{k}{\Rightarrow} w$ — By inv. of E-OPEN-$\star$ on 7.63. (7.64)

$\text{fa}_{\text{env}}(\Gamma, \rho) \vdash_{\text{suff-disj}} v_{\text{obj}} : \tau_{\text{obj}}$ — Likewise. (7.65)

$\forall i.\ \exists v_i.\ \Gamma_{\text{IH}} \vdash_{\text{exe}} \langle e, \rho'\,[x_i \to v'_i]_i\rangle \overset{k'}{\Rightarrow} w'$ — Likewise. (7.66)

$\text{fa}_{\text{env}}(\Gamma, \rho') \vdash_{\text{suff-disj}} v'_{\text{obj}} : \tau_{\text{obj}}$ — Likewise. (7.67)

$v_{\text{obj}} \triangleq \textbf{prod}\,(v_i)_i =_\alpha \rho(x_{\text{obj}}) : \tau_{\text{obj}}$
$v'_{\text{obj}} \triangleq \textbf{prod}\,(v'_i)_i =_\alpha \rho'(x_{\text{obj}}) : \tau_{\text{obj}}$ — Likewise. (7.68)

$\sigma(\rho) =_\alpha \sigma'(\rho') : \Gamma$ — By hypothesis . (7.69)

And,

$\sigma(v_{\text{obj}}) =_\alpha \sigma(\rho(x_{\text{obj}})) : \tau_{\text{obj}}$ — By 7.68 and Lemma 7.35. (7.70)

$=_\alpha \sigma'(\rho'(x_{\text{obj}})) : \tau_{\text{obj}}$ — By 7.69.

$=_\alpha \sigma'(v'_{\text{obj}}) : \tau_{\text{obj}}$ — By 7.68 and Lemma 7.35. (7.71)

$\sigma(v_{\text{obj}}) =_\alpha \sigma'(v'_{\text{obj}}) : \tau_{\text{obj}}$ — By 7.70–7.71 and Lemma 7.16. (7.72)

And,

$\sigma(\text{fa}_{\text{env}}(\Gamma, \rho)) \vdash_{\text{suff-disj}} \sigma(v_{\text{obj}}) : \tau_{\text{obj}}$
$\sigma'(\text{fa}_{\text{env}}(\Gamma, \rho')) \vdash_{\text{suff-disj}} \sigma'(v'_{\text{obj}}) : \tau_{\text{obj}}$ — By 7.65 and 7.67 and Lemma 7.49. (7.73)

$\sigma(\text{fa}(\tau_{\text{obj}}, v_{\text{obj}})) \vdash_{\text{suff-disj}} \sigma(v_{\text{obj}}) : \tau_{\text{obj}}$
$\sigma'(\text{fa}(\tau_{\text{obj}}, v_{\text{obj}})) \vdash_{\text{suff-disj}} \sigma'(v'_{\text{obj}}) : \tau_{\text{obj}}$ — By 7.70 and 7.71 and Lemma 7.36.

https://doi.org/10.1017/S0956796816000137 Published online by Cambridge University Press

$\exists \sigma_{\text{op}}, \sigma'_{\text{op}}.$

$\text{dom}(\sigma_{\text{op}}) \subseteq \text{xa}(\tau_{\text{obj}}, \sigma(v_{\text{obj}}))$
$\text{dom}(\sigma'_{\text{op}}) \subseteq \text{xa}(\tau_{\text{obj}}, \sigma'(v'_{\text{obj}}))$ — By 7.65 on Lemma 7.42. (7.74)

Such that

$\forall i.\ \sigma_{\text{op}}(\sigma(v_i)) =_\alpha \sigma'_{\text{op}}(\sigma'(v'_i)) : \tau$ — Likewise. (7.75)

$\text{rng}(\sigma_{\text{op}}), \text{rng}(\sigma'_{\text{op}}) \# \sigma, \sigma', \rho, \rho', w, w', v_{\text{obj}}, v'_{\text{obj}}$ — Likewise. (7.76)

$\text{inj}(\sigma_{\text{op}})$ and $\text{inj}(\sigma'_{\text{op}})$ — Likewise. (7.77)

And,

$\text{dom}(\sigma_{\text{op}}) \# \text{rng}(\sigma_{\text{op}})$
$\text{dom}(\sigma'_{\text{op}}) \# \text{rng}(\sigma'_{\text{op}})$ — By 7.74 and 7.76 and Lemma 7.2. (7.78)

Define

$\sigma_{\text{IH}} \triangleq \sigma_{\text{op}} \circ \sigma$ and $\sigma'_{\text{IH}} \triangleq \sigma'_{\text{op}} \circ \sigma'$

Now,

$\text{xa}(\tau_{\text{obj}}, \sigma(v_{\text{obj}})) \# \text{fa}_{\text{env}}(\Gamma, \sigma(\rho))$
$\text{xa}(\tau_{\text{obj}}, \sigma'(v'_{\text{obj}})) \# \text{fa}_{\text{env}}(\Gamma, \sigma'(\rho'))$ — By 7.73 and def. of $\vdash_{\text{suff-disj}}$ and Lemma 7.26. (7.79)

$\sigma_{\text{IH}}(\rho) =_\alpha \sigma'_{\text{IH}}(\rho') : \Gamma$ — By 7.69, 7.74, and 7.78 on Lemma 7.31.

$\sigma_{\text{IH}}(\rho\,[x_i \to v_i]_i) =_\alpha \sigma'_{\text{IH}}(\rho'\,[x_i \to v'_i]_i) : \Gamma_{\text{IH}}$ — By 7.75. (7.80)

And,

$\text{inj}(\sigma_{\text{IH}})$ and $\text{inj}(\sigma'_{\text{IH}})$ — Because $\text{inj}(\sigma)$ and $\text{inj}(\sigma')$ and by 7.76 and 7.77. (7.81)

And,

$\text{rng}(\sigma_{\text{IH}}) \# \rho\,[x_i \to v_i]_i$
$\text{rng}(\sigma'_{\text{IH}}) \# \rho'\,[x_i \to v'_i]_i$ — Because $\text{rng}(\sigma) \# \rho$ and $\text{rng}(\sigma') \# \rho'$ and by 7.76.

So:

$\sigma_{\text{IH}}(w) =_\alpha \sigma'_{\text{IH}}(w') : \tau$ — By 7.80 and 7.81 on IH. (7.82)

Case 1:

$w = \text{FAULT}$ or $w' = \text{FAULT}$

$w = w' = \text{FAULT}$ — By 7.82 on Lemma 7.46. (7.83)

$w_{\text{out}} = w'_{\text{out}} = \text{FAULT}$ — By 7.64 and 7.66 and E-OPEN-$\star$.

Case 2:

$w \neq \text{FAULT}$ and $w' \neq \text{FAULT}$ (7.84)

https://doi.org/10.1017/S0956796816000137 Published online by Cambridge University Press

We know,

$\sigma_{\mathrm{IH}}(v_{\mathrm{obj}}) =_\alpha \sigma_{\mathrm{IH}}(v'_{\mathrm{obj}}) : \tau_{\mathrm{obj}}$ — By 7.75 on Lemma 7.45.

$\mathrm{xa}(\tau_{\mathrm{obj}}, \sigma_{\mathrm{IH}}(v_{\mathrm{obj}})) = \mathrm{xa}(\tau_{\mathrm{obj}}, \sigma'_{\mathrm{IH}}(v'_{\mathrm{obj}}))$ — By def. of xa and 7.75 and Lemma 7.36. (7.85)

$\sigma_{\mathrm{IH}}(\mathrm{xa}(\tau_{\mathrm{obj}}, v_{\mathrm{obj}})) = \sigma'_{\mathrm{IH}}(\mathrm{xa}(\tau_{\mathrm{obj}}, v'_{\mathrm{obj}}))$ — By 7.76 and 7.81 and Lemma 7.27. (7.86)

And,

$\mathrm{fa}(\tau, \sigma_{\mathrm{IH}}(w)) = \mathrm{fa}(\tau, \sigma'_{\mathrm{IH}}(w'))$ — By 7.82 on Lemma 7.36.

$\sigma_{\mathrm{IH}}(\mathrm{fa}(\tau, w)) = \sigma'_{\mathrm{IH}}(\mathrm{fa}(\tau, w'))$ — By 7.81 on Lemma 7.26.

$\sigma_{\mathrm{IH}}(\mathrm{xa}(\tau_{\mathrm{obj}}, v_{\mathrm{obj}})) \# \sigma_{\mathrm{IH}}(\mathrm{fa}(\tau, w))$
iff $\sigma'_{\mathrm{IH}}(\mathrm{xa}(\tau_{\mathrm{obj}}, v'_{\mathrm{obj}})) \# \sigma'_{\mathrm{IH}}(\mathrm{fa}(\tau, w'))$ — By 7.86.

$\mathrm{xa}(\tau_{\mathrm{obj}}, v_{\mathrm{obj}}) \# \mathrm{fa}(\tau, w)$
iff $\mathrm{xa}(\tau_{\mathrm{obj}}, v'_{\mathrm{obj}}) \# \mathrm{fa}(\tau, w')$ — By Lemma 7.1.

So, case 2.1:

$\mathrm{xa}(\tau_{\mathrm{obj}}, v_{\mathrm{obj}}) \# \mathrm{fa}(\tau, w)$
and $\mathrm{xa}(\tau_{\mathrm{obj}}, v'_{\mathrm{obj}}) \# \mathrm{fa}(\tau, w')$ (7.87)

$w_{\mathrm{out}} = w$ and $w'_{\mathrm{out}} = w'$ — By 7.64 and 7.66 and E-Open-Ok. (7.88)

And also,

$\sigma(\mathrm{xa}(\tau_{\mathrm{obj}}, v_{\mathrm{obj}})) \# \sigma(\mathrm{fa}(\tau, w))$
$\sigma'(\mathrm{xa}(\tau_{\mathrm{obj}}, v'_{\mathrm{obj}})) \# \sigma'(\mathrm{fa}(\tau, w'))$ — By 7.87 and because inj($\sigma$).

$\mathrm{dom}(\sigma_{\mathrm{op}}) \# \sigma(\mathrm{fa}(\tau, w))$
$\mathrm{dom}(\sigma'_{\mathrm{op}}) \# \sigma'(\mathrm{fa}(\tau, w'))$ — By 7.74 and Lemma 7.27.

$\mathrm{dom}(\sigma_{\mathrm{op}}) \# \mathrm{fa}(\tau, \sigma(w))$
$\mathrm{dom}(\sigma'_{\mathrm{op}}) \# \mathrm{fa}(\tau, \sigma'(w'))$ — By Lemma 7.26.

$\sigma_{\mathrm{IH}}(w) =_\alpha \sigma(w) : \tau$
$\sigma'_{\mathrm{IH}}(w') =_\alpha \sigma'(w') : \tau$ — By 7.77 and 7.78 on Lemma 7.31.

$\sigma(w) =_\alpha \sigma'(w') : \tau$ — By 7.82 and Lemma 7.16.

$\sigma(w_{\mathrm{out}}) =_\alpha \sigma'(w'_{\mathrm{out}}) : \tau$ — By 7.88.

Or, case 2.2:

$w_{\mathrm{out}} = \textsc{fault}$ and $w'_{\mathrm{out}} = \textsc{fault}$ — By 7.64 and 7.66 and E-Open-Ok.

https://doi.org/10.1017/S0956796816000137 Published online by Cambridge University Press

- *Case* E-IF-⋆:

  First,

  | | |
  |---|---|
  | $\rho(x_0) = a$ and $\rho'(x_0) = a'$<br>$\rho(x_0) = b$ and $\rho'(x_0) = b'$ | By inv. of E-IF-⋆. |
  | $\sigma(a) =_\alpha \sigma'(a') : \tau'$ and $\sigma(b) =_\alpha \sigma'(b') : \tau'$ | Because $\sigma(\rho) =_\alpha \sigma'(\rho') : \Gamma$.. |
  | $\sigma(a) = \sigma'(a')$ and $\sigma(b) = \sigma'(b')$ | By Rα-RATOM or Bα-BATOM. |
  | $\sigma(a) = \sigma(b)$ iff $\sigma'(a') = \sigma'(b')$ | Trivially. |
  | $a = b$ iff $a' = b'$ | Because inj($\sigma$). |

  Without loss of generality, assume

  $$a = b \tag{7.89}$$

  Now,

  | | |
  |---|---|
  | $\Gamma \vdash_{\text{exe}} \langle e_0, \rho \rangle \overset{k}{\Rightarrow} w$ | By inv. of E-IF-⋆. (7.90) |
  | $\Gamma \vdash_{\text{exe}} \langle e_0, \rho' \rangle \overset{k'}{\Rightarrow} w'$ | Likewise. |
  | $w =_\alpha w' : \tau$ | By 7.89 and 7.90 and because $\sigma(\rho) =_\alpha \sigma'(\rho') : \Gamma$ on IH. |

The cases for E-CALL, E-LET-⋆, E-CASE-⋆, and E-QLIT are available in the supplementary material. □

*Theorem 7.2* (*α-equivalent environments have equivalent termination behavior*)

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

*Proof*
By induction on $k$, and case analysis on $e$:

- *Case* E-FRESH-OK: (**fresh** $x$ **in** $e$)
  It suffices to show lines 7.95 and 7.94.

  First,

  | | |
  |---|---|
  | $\Gamma, x{:}\text{BAtom} \vdash_{\text{exe}} \langle e, \rho\,[x \to a] \rangle \overset{k}{\Rightarrow} w$ | By inv. of E-FRESH-OK. (7.91) |
  | $a \notin \text{fa}_{\text{env}}(\Gamma, \rho)$ and $w = \text{FAULT} \vee a \notin \text{fa}(\tau, w)$ | Likewise. (7.92) |

  But

  | | |
  |---|---|
  | $\rho\,[x \to a] =_\alpha \rho'\,[x \to a] : \Gamma, x{:}\text{BAtom}$ | Because $\rho =_\alpha \rho' : \Gamma$. |
  | $\exists w'.\ \Gamma, x{:}\text{BAtom} \vdash_{\text{exe}} \langle e, \rho'\,[x \to a] \rangle \overset{k}{\Rightarrow} w'$ | By 7.91 and IH. (7.93) |

https://doi.org/10.1017/S0956796816000137 Published online by Cambridge University Press

$w =_\alpha w' : \tau$ Likewise. (7.94)

But

$a \notin \mathrm{fa}_{\mathrm{env}}(\Gamma, \rho')$ and $w' = \textsc{fault} \vee a \notin \mathrm{fa}(\tau, w')$ By 7.92 on Lemma 7.36.

$\Gamma \vdash_{\mathrm{exe}} \langle (\textbf{fresh } x \textbf{ in } e), \rho' \rangle \overset{k+1}{\Longrightarrow} w'$ By E-Fresh-Ok. (7.95)

- *Case* E-Fresh-Fail: This case proceeds almost identically to E-Fresh-Ok.
- *Case* E-Open-Ok: (**open** $x$ $((x_i)_i)$ $e$)
  It suffices to show lines 7.102 and 7.100.

  First,

  $\Gamma, (x_i{:}\tau_i)_i \vdash_{\mathrm{exe}} \langle e, \rho\,[x_i \to w_i]_i \rangle \overset{k}{\Rightarrow} w$ By inv. of E-Open-Ok. (7.96)

  $\mathrm{fa}_{\mathrm{env}}(\Gamma, \rho) \vdash_{\text{suff-disj}} \textbf{prod}\,(w_i)_i : \tau_{\mathrm{obj}}$ Likewise. (7.97)

  $w = \textsc{fault} \vee \mathrm{xa}(\tau_{\mathrm{obj}}, \textbf{prod}\,(w_i)_i) \# \mathrm{fa}(\tau, w)$ Likewise. (7.98)

  But

  $\rho\,[x_i \to w_i]_i =_\alpha \rho'\,[x_i \to w_i]_i : \Gamma, (x_i{:}\tau_i)_i$ Because $\rho =_\alpha \rho' : \Gamma$ and by Lemma 7.14.

  $\exists w'.\ \Gamma', (x_i{:}\tau_i)_i \vdash_{\mathrm{exe}} \langle e, \rho'\,[x_i \to w_i]_i \rangle \overset{k}{\Rightarrow} w'$ By 7.96 and IH. (7.99)

  $w =_\alpha w' : \tau$ Likewise. (7.100)

  But

  $\mathrm{fa}_{\mathrm{env}}(\Gamma, \rho) = \mathrm{fa}_{\mathrm{env}}(\Gamma, \rho')$ Because $\rho =_\alpha \rho' : \Gamma$ and by Lemma 7.36.

  $\mathrm{fa}_{\mathrm{env}}(\Gamma, \rho') \vdash_{\text{suff-disj}} \textbf{prod}\,(w_i)_i : \tau_{\mathrm{obj}}$ By 7.97. (7.101)

  And,

  $\mathrm{fa}(\tau, w) = \mathrm{fa}(\tau, w')$ By 7.100 on Lemma 7.36.

  And,

  $w' = \textsc{fault} \vee \mathrm{xa}(\tau_{\mathrm{obj}}, \textbf{prod}\,(w'_i)_i) \# \mathrm{fa}(\tau, w')$ By 7.98 and 7.100.

  $\Gamma \vdash_{\mathrm{exe}} \langle (\textbf{open } x\ ((x_i)_i)\ e), \rho' \rangle \overset{k+1}{\Longrightarrow} w'$ By E-Open-Ok on 7.101 and 7.99. (7.102)

- *Case* E-Open-Fail: This case proceeds almost identically to E-Open-Ok.

The cases for E-Call, E-If-$\star$, E-QLit, E-Case-$\star$, and E-QLit are available in the supplementary material. □

## 8 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 8.1).

The proof system's judgment is of the form $\Gamma \vdash_{\mathrm{proof}} \{H\}\ e\ \{P\}$. 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.

https://doi.org/10.1017/S0956796816000137 Published online by Cambridge University Press

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 post-condition, contains predicates that describe the connection between atoms in the environment and atoms in the output. $C$ is used for other constraints.

The obligations emitted by the deduction system must be satisfied by showing them true for all $\rho$ compatible with $\Gamma$; in practice, we do this with an SMT (satisfiability modulo theories) Solver. (For example, Romeo-L (Muehlboeck, 2013) uses Z3 (de Moura & Bjørner, 2008). We discuss this in more detail in Section 9.8.)

We begin by giving the syntax of constraints.

$$
\begin{aligned}
z \in \mathit{ConstrSetVar} &::= x \mid \cdot \\
s \in \mathit{SetDesc} &::= \varnothing \\
&\mid\ s \cup s \\
&\mid\ s \cap s \\
&\mid\ \mathit{sf}(z) \\
&\mid\ \mathscr{F}_e(\Gamma) \\
&\mid\ \mathscr{X}_\tau(x_i)_i \\
\mathit{sf} \in \mathit{SetFn} &::= \mathscr{F} \mid \mathscr{F}_r \mid \mathscr{F}_b \\
H, P, C \in \mathit{Constraint} &::= C \wedge C \\
&\mid\ s = s \\
&\mid\ s \neq s \\
&\mid\ s \# s \\
&\mid\ s \subseteq s \\
&\mid\ z \cong e^{\mathrm{qlit}} \\
&\mid\ \textbf{true} \\
\Gamma_{\mathrm{dot}} \in \mathit{TypeEnvWithDot} &::= \epsilon \mid \Gamma_{\mathrm{dot}}, z{:}\tau
\end{aligned}
$$

Formulas are constructed from variables $z$, which range over program variables $x$, and $\cdot$ (which refers to the output value of the current expression). Set-valued terms are constructed from the free names ($\mathscr{F}$), free references ($\mathscr{F}_r$), free binders ($\mathscr{F}_b$), and exposable names ($\mathscr{X}$) of values (here, the corresponding types can be retrieved from the environment), and the free names ($\mathscr{F}_e$) of environments (which is syntax sugar for a union of $\mathscr{F}$s), and then by the standard set constructors ($\cup$ and $\cap$). Atomic formulas denote equality, inequality, etc., of sets, and $\cong$ denotes that two values have the same free binders and references. Finally, constraints are conjunctions of atomic formulas.

The syntax for the $\mathscr{X}$ set function is unusal; it consumes the subterms of its "argument" instead of the argument itself, and also a type indicating its binding structure. (The set functions that take a single variable as an argument do not need this, because a type environment is available.) This is because, unlike $\mathscr{F}$, $\mathscr{F}_r$, and $\mathscr{F}_b$, the result of $\mathscr{X}$ depends on bound names of its argument. However, it only depends on the free names of the *subterms* of its argument. This way, two values that are $\cong$ have identical behavior under all the set functions. Furthermore, this syntax fits the way that $\mathscr{X}$ is used in P-Open.

We use quasi-literals to describe values with variable interpolation. Because the type environment $\Gamma$ is present, the type annotations of quasi-literals are redundant,

https://doi.org/10.1017/S0956796816000137 Published online by Cambridge University Press

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

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

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

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

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

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

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

$$\frac{(x_i{:}\tau_i)_i \vdash_{\text{proof}} \{C_0\}\ e\ \{C_1\}}{\vdash_{\text{proof}} (\textbf{define-fn}\ (f\ (x_i{:}\tau_i)_i\ \textbf{pre}\ C_0) : \tau_0\ e\ \textbf{post}\ C_1)\ \text{ok}}\ \text{P-FNDEF}$$

$$\frac{\forall i.\ \vdash fD_i\ \text{ok} \qquad \varepsilon \vdash_{\text{proof}} \{\textbf{true}\}\ e\ \{\textbf{true}\}}{\vdash_{\text{proof}} (fD_i)_i\ e\ \text{ok}}\ \text{P-PROG}$$

Fig. 6. Verification rules for the deduction system.

but for economy of abstraction, we elected to reuse an existing concept instead of creating a new one.

In general, our rules, in Figure 6, are patterned after those in Pottier (2007b), using the type information in $\Gamma$ to collect information about values. This subsumes Pottier's $\Delta$. 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 $\Gamma_{\text{dot}} \vDash H \Rightarrow P$. P-QLIT has only one obligation, which is to ensure that the result it produces 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 pre-condition is true (given $H$), and second that the resulting value satisfies the constraints in $P$ (given $H$ and the post-condition 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.

https://doi.org/10.1017/S0956796816000137 Published online by Cambridge University Press

Proving the theorems in Section 6 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 from two different binding relationships in the same environment (thereby purporting to show equality between two names that are not related by binding). For the purposes of dynamically respecting $\alpha$-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 *post-condition* 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 using $\cong$. This information conveys the relationship between the atoms in the scrutinee ($x_{\text{obj}}$ and $x_0$, respectively) and the atoms in its component(s).

### *8.1 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 $\cdot$.

A similar issue occurs in P-CALL. The pre- and post-conditions of the function (not to be confused with the expression's logical post-condition $P$) 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 among Romeo program variables is incompatible with the deduction system, because obligations must be able to refer to (and distinguish) everything in $\Gamma$ by name. This gives rise to the requirement that certain $x$'s be fresh for $\Gamma$, $H$, and $P$; this requirement is easily satisfied by a simple renaming pass prior to type and proof checking.

P-CALL's body hypothesis contains $\mathscr{F}(\cdot) \subseteq \mathscr{F}_{\text{e}}\left(\left(x_{\text{actual},i}{:}\tau_{\text{formal},i}\right)_i\right)$, a term representing extra information as a consequence of Lemma 8.1, which states that the free atoms in the result of any expression are a subset of the free atoms in the environment in which it is evaluated. A similar term appears in the hypothesis for P-LET's body subexpression.

https://doi.org/10.1017/S0956796816000137 Published online by Cambridge University Press

Finally, in P-IfEq, the information resulting from 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.

### 8.2 Soundness of the deduction system

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

*Theorem 8.1* (*Soundness of the deduction system*)

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

*Proof*
By induction on $k$. □

The most important lemma for proving Theorem 8.1 is "No Names Made Up", which is similar to Pottier's No Atoms Made Up lemma (2007a).

*Lemma 8.1* (*No names made up*)

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

*Proof*
By induction on $k$. □

The proofs of Lemma 8.1 and Theorem 8.1 can be found in the supplementary material.

### 8.3 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**, does not have *bv* free: its left-hand child, *bv*, is an unexported binder, and its right-hand child imports it, meaning that *bv* can't be free in it either, regardless of what *convert* returns.

https://doi.org/10.1017/S0956796816000137 Published online by Cambridge University Press

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 as free variables. 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 lines 13–14. We first examine its left-hand side, which is a lambda binding *bv* in *e-rest*. Therefore, by a similar argument to the one above, *bv* is no problem, and we only 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 function application 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 8.1, 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 (Pottier, 2007b), and the same examples apply.

## 9 Related work

### *9.1 Statically specified binding in template macros*

The work of Herman and Wand (Herman & Wand, 2008; Herman, 2010) introduced the idea of a static binding specification for a template-based 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 of these is that where Herman and Wand use addresses into binary trees of values, we use indices into wide products. However, this does not affect the representational power of the system.

### *9.2 Pure FreshML*

The second key source for this work is Pure FreshML (Pottier, 2007b). 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

https://doi.org/10.1017/S0956796816000137 Published online by Cambridge University Press

proof system that generates proof obligations which, if true, guarantee statically that no faults will occur. One difference, important for application to 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**).

Pure FreshML leaves the actual language of binding specifications underdetermined. All the formal development is done in a simple system, roughly equivalent to the $\lambda$-calculus, but one of the key examples, normalization by evaluation, is done using the more expressive system of C$\alpha$ml (Pottier, 2006). 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.

Pure FreshML's safety guarantees are proved true in a nominal logic in which values are entire $\alpha$-equivalence classes. However, we know of no nominal logic powerful enough to represent the complex binding structures available in Romeo, so our proof of correctness must do the work of a nominal logic "by hand".

### *9.3 Ott*

Ott (Sewell *et al*., 2010) 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 capture-avoiding substitution in 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 $\alpha$-equivalence between its "concrete abstract syntax trees" (the equivalent of our values $v$). This 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 $\alpha$-equivalence: two trees are $\alpha$-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 7.1 and 7.2.

### *9.4 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.

Despite extensive use of hygienic macro systems, what it meant for them to be "hygienic" has resisted definition until recently. The algorithm described by Dybvig

https://doi.org/10.1017/S0956796816000137 Published online by Cambridge University Press

(1992), 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.

More recently, a reformulation by Matthew Flatt (2016) called "sets of scopes" provides largely the same behavior with a simpler semantics and in an easier-to-use fashion.

The original attempts at specification, given by Clinger (1991) and Dybvig (1992), 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 is circular. Recent work by Adams (2015) presents a plausible definition that examines the support of unexpanded code, while being agnostic as to its binding structure. However, we feel that the use of static binding specifications (introduced for this purpose by Herman (2010)) provides a more straightforward rigorous basis for hygiene, at least in situations in which binding specifications are appropriate.

Secondly, the Dybvig algorithm offers no *static* guarantees. If the macro 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.

### 9.5 Binding in theorem-proving systems

Ever since the POPLMark Challenge (Aydemir *et al.*, 2005), there has been a large interest in coding terms with bindings in various proof assistants (Urban, 2008; Pouillard & Pottier, 2010; Pollack *et al.*, 2012). 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).

### 9.6 Unbound

Unbound (Weirich *et al.*, 2011) is a Haskell library for safely manipulating syntax that operates on many of the same principles as the FreshML family that Romeo comes from. In particular, Unbound's **unbind** function corresponds roughly to our **open** form, and **bind** *p e* corresponds to a quasiliteral of the form $\left(\textbf{prod}\ e_{\mathrm{p}}^{\mathrm{qlit}}, e_{\mathrm{e}}^{\mathrm{qlit}}{\downarrow}0\right)$.

Although it has a number of internal differences (such as the locally nameless representation of syntax), the programming model is fairly similar. The primary difference noticeable by the user is Unbound's binding specification system, which is similar to C$\alpha$ml (Pottier, 2006), but extended to the point that it supports constructs like `let*`. Like C$\alpha$ml, it is oriented toward abstract syntax, not concrete syntax like Romeo.

https://doi.org/10.1017/S0956796816000137 Published online by Cambridge University Press

### 9.7 SoundX

In general, it is impossible to typecheck an expression without knowing its type environment. Historically, this has been the reason that languages with both a type system and a macro system perform typechecking after macro expansion. The unfortunate consequence of this is that the user is presented type errors in terms of code that they didn't even write.

SoundX (Lorenzen & Erdweg, 2016) is a step toward a solution to this problem, demonstrating extensions to language syntax which are less powerful than macros, but whose expansions are guaranteed to be type-safe. We believe that the major remaining steps between SoundX and true typed macros are:

- Extensions defined by arbitrary code, instead of pattern matching. This is essentially the difference between $\lambda_m$ (Herman, 2010) and Romeo. Proof that syntax extensions preserve the validity of type judgments must be providable by the typesystem, which might be very complex.
- Locally scoped extensions. This is potentially complex, but it may be able to at least follow the example from hygienic locally scoped macros in Scheme.
- User-friendly binding safety. SoundX extensions are statically rejected if they might perform accidental capture, requiring manual freshening and contortions around capturing constructs, and restricting what names the extension's user may choose.[6] We have shown that it is instead possible to automatically rename to avoid collisions, which is far more convenient to the user.

We believe that typed macros will be a major user-friendliness improvement for the interaction between types and macros.

### 9.8 Romeo-L

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, not more complicated expressions.

A second problem is that we have not yet described how the truth of $\Gamma \models H \Rightarrow P$ 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 (2013), 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 a constraint $C$. Therefore, the user need only supply constraints for function definitions. In practice, this means a vastly reduced annotation burden.

[6] Although it is not immediately obvious, SoundX does posess binding specifications. This is by virtue of its type rules. For example, typing a $\lambda$ expression in the environment $\Gamma$ requires typing its body in the environment $\Gamma, x : \tau$, where $x$ is the name bound by the lambda, and $\tau$ is its type. This indicates that $x$ is bound in the $\lambda$'s body.

https://doi.org/10.1017/S0956796816000137 Published online by Cambridge University Press

Romeo-L also includes a connection to the Z3 SMT solver (de Moura & Bjørner, 2008), which is able to check statements of the form $\Gamma \models H \Rightarrow P$, completing the automated checking of the deduction system. The translation takes the constraints $H$ and $P$, and (in a sense) partially evaluates them until they are expressed only in terms of the free binders and free references of program variables that are never destructured (if a variable is destructured, the sets of atoms relevant to it can be expressed in terms of the variables it is destructured into). The sets of atoms that remain are uninterpreted, except to add size constraint approximations (when it is possible to determine that the number of elements of a set is 0, 1, $\leqslant 1$, or $\geqslant 1$). At that point, the constraints are directly expressible as SMT problems using combinatory array logic as the concrete theory.

Furthermore, Romeo-L 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**.

## Acknowledgments

We would like to thank Fabian Muehlboeck for his work in testing and validating Romeo, and Dionna Glaze for her collaboration in hammering out the formal properties of our binding system. We would also like to thank the anonymous reviewers for their comments to improve this paper.

## References

Adams, M. D. (2015) Towards the essence of hygiene. In Proceedings of the 42Nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. New York, NY, USA: ACM, pp. 457–469.

Aydemir, B. E., Bohannon, A., Fairbairn, M., Foster, J. N., Pierce, B. C., Sewell, P., Vytiniotis, D., Washburn, G., Weirich, S. & Zdancewic, S. (2005). Mechanized metatheory for the masses: The PoplMark challenge. In Proceedings of the 18th International Conference on Theorem Proving in Higher Order Logics. Berlin, Heidelberg: Springer-Verlag, pp. 50–65.

Clinger, W. & Rees, J. (1991). Macros that work. In Proceedings of the 18th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. New York, NY, USA: ACM, pp. 155–162.

de Moura, L. & Bjørner, N. (2008). Z3: An efficient SMT solver. In Tools and Algorithms for the Construction and Analysis of Systems. Lecture Notes in Computer Science, vol. 4963, pp. 337–340. Berlin Heidelberg: Springer.

Dybvig, R. K., Hieb, R. & Bruggeman, C. (1992). Syntactic abstraction in scheme. *LISP Symb. Comput.* **5**(4), 295–326.

Erdweg, S., van der Storm, T. & Dai, Y. (2014). Capture-avoiding and hygienic program transformations. In *ECOOP 2014 Object-Oriented Programming*, Jones, R. (ed), Lecture Notes in Computer Science, vol. 8586, pp. 489–514. Berlin Heidelberg: Springer.

Flatt, M. (2016). Binding as sets of scopes. *SIGPLAN Not.* **51**(1), 705–717.

Hendriks, D. & van Oostrom, V. (2003). Adbmal. In *Automated Deduction CADE-19*, Baader, F. (ed), Lecture Notes in Computer Science, vol. 2741. Berlin Heidelberg: Springer, pp. 136–150.

Herman, D. (2010). *A Theory of Typed Hygienic Macros*. PhD Thesis, Boston, MA, USA: Northeastern University.

Herman, D. & Wand, M. (2008). A theory of hygienic macros. In Proceedings of the Theory and Practice of Software, 17th European Conference on Programming Languages and Systems. Berlin, Heidelberg: Springer-Verlag, pp. 48–62.

Kohlbecker, E., Friedman, D. P., Felleisen, M. & Duba, B. (1986). Hygienic macro expansion. In Proceedings of the 1986 ACM Conference on Lisp and Functional Programming, LFP '86. New York, NY, USA: ACM, pp. 151–161.

Lorenzen, F. & Erdweg, S. (2016). Sound type-dependent syntactic language extension. In Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016. New York, NY, USA: ACM.

Muehlboeck, F. (2013). *Checking Binding Hygiene Statically*. Master's Thesis, Boston, MA, USA: Northeastern University.

Pollack, R., Sato, M. & Ricciotti, W. (2012). A canonical locally named representation of binding. *J. Autom. Reason.* **49**(2), 185–207.

Pottier, F. (2006). An overview of Cαml. *Electron. Notes Theor. Comput. Sci.* **148**(2), 27–52.

Pottier, F. (2007a). *Static Name Control for* FreshML. Available at: `http://gallium.inria.fr/~fpottier/publis/fpottier-pure-freshml-long.pdf`. [4 July, 2016]

Pottier, F. (2007b). Static name control for FreshML. In Proceedings of the 22nd Annual IEEE Symposium on Logic in Computer Science. Washington, DC, USA: IEEE Computer Society, pp. 356–365.

Pouillard, N. & Pottier, F. (2010). A fresh look at programming with names and binders. In Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming. New York, NY, USA: ACM, pp. 217–228.

Sewell, P., Nardelli, F. Z., Owens, S., Peskine, G., Ridge, T., Sarkar, S. & Strnia, R. (2010). Ott: Effective tool support for the working semanticist. *J. Funct. Program.* **20**, 71–122.

Shinwell, M. R., Pitts, A. M. & Gabbay, M. J. (2003). FreshML: Programming with binders made simple. In Proceedings of the 8th ACM SIGPLAN International Conference on Functional Programming, ICFP '03, pp. 263–274. New York, NY, USA: ACM.

Urban, C. (2008). Nominal techniques in Isabelle/HOL. *J. Autom. Reason.* **40**(4), 327–356.

Weirich, S., Yorgey, B. A. & Sheard, T. (2011). Binders unbound. In Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming. New York, NY, USA: ACM, pp. 333–345.