THESIS TITLE: Flexible Binding-Safe Programming

AUTHOR: Paul Stansifer

Ph.D. Thesis Approved to complete all degree requirements for the Ph.D. Degree in Computer Science.

Thesis Advisor

Date 4/6/16

Thesis Reader

Date 1/6/16

Thesis Reader

Date 2016/4/6

Thesis Reader

Date

FINPLER

Date April 6, 2016

GRADUATE SCHOOL APPROVAL:

Director, Graduate School

Date APRIL 22, 2016

COPY RECEIVED IN GRADUATE SCHOOL OFFICE:

Recipient's Signature

Date 1/27/16

Distribution: Once completed, this form should be scanned and attached to the front of the electronic dissertation document (page 1). An electronic version of the document can then be uploaded to the Northeastern University-UMI website.
Flexible binding-safe programming

Paul Stansifer

April, 2016
Abstract

Current nominal systems for safely manipulating values with names, like Pure FreshML, 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 by applying those nominal techniques to a richer specification system, inspired by attribute grammars. Our system has the expressive power of David Herman’s $\lambda_m$, but is a full-fledged programming system for any kind of metaprogramming.

We demonstrate our system first by implementing it in a core calculus we call Romeo, and which we prove takes $\alpha$-equivalent inputs to $\alpha$-equivalent outputs. Then we use the same mechanics in PLT Redex to provide similar safety guarantees in that context, and demonstrate the ease of retrofitting existing Redex models to take advantage of our binding specifications.
Acknowledgments

This dissertation is the distillation of the work of many people. I would like to take a brief break from the first person plural to thank them.

Mitch Wand has been my advisor since I arrived at Northeastern. I tremendously appreciate his patience and good humor, but more important for my growth as a computer scientist has been his ability to never answer a question for me if I could answer it myself.

My work on Redex would have been impossible without the help of Robby Findler. He is passionate about making Redex as good a tool as it can be, and working with him on that goal has been a pleasure.

Amal Ahmed’s rigorous mathematical reading of this work and her insights about how to present technical material greatly improved the final version.

Olin Shivers and Amal both encouraged me to address the topic of typed macro systems in the final chapter. Although the topic doesn’t occupy many words, thinking about this subject directly has significantly changed how I see my own research.

Matthias Felleisen secured funding that enabled me to finish this dissertation at an appropriate pace, instead of rushing it out the door.

Dionna Glaze was an early collaborator on the design of Romeo, and spent a great deal of time hammering out the details of the binding system with me. I am also grateful for the tautological yet aesthetically-pleasing discovery that $\bot \vdash \top$.

Fabian Muehlboeck’s work on Romeo-L was tremendously important to me. Error messages are, in a sense, the entirety of the user interface of a programming language, and Fabian’s success at turning an SMT solver’s counterexample into a useful error message is a research discovery that makes me excited and hopeful for the future.

To the anonymous reviewers who read the papers that turned into this dissertation, I want to express my gratitude for reading these words when they were not as good, and helping to make them better.

There are some names that can’t not appear here: Sean Carollo, Julianne Shelby, Jennifer James, and Christine Vaughn. Thank you for listening to me, and thank you for having such interesting things to say.

Although this work (hopefully) has other reasons to exist, I think that a major subconscious part of my desire to pursue a Ph.D. was my admiration for my parents, Ryan and Betsy. I don’t know whether it is a coincidence that my work also is about languages, but their love for all forms of learning has always been an inspiration to me.

My brother Eric has a fine sense of the ridiculous; an important characteristic for a coder and a mathematician. I leaned on him a lot when the absurdity inherent in the act of programming was too much to bear alone, or when I needed to tell someone the real name of a certain lemma.

It’s appallingly traditional to end the acknowledgments by thanking one’s spouse for their support. However, for all of her technical contributions (editing rough prose, late-night proofreading, and evaluating technical explanations), I am still more grateful to Katie for believing that the symbols on these pages actually matter.
# Contents

1 Introduction .............................................. 5  
1.1 $\lambda_m$ .............................................. 6  
1.2 Pure FreshML ........................................... 6  
1.3 Structure of this work ................................ 7  
1.3.1 Romeo ............................................... 7  
1.3.2 Redex with binding specifications ................. 8  

2 Binding language ........................................ 10  
2.1 Overview of binding types ............................... 10  
2.1.1 Example: multiple, partially-shared bindings ...... 11  
2.2 Binding types, in more detail ........................... 12  
2.3 The algebra of binding combinators .................... 13  

3 $\alpha$-equivalence ......................................... 15  
3.1 Binder equivalence ...................................... 15  
3.2 Reference equivalence .................................. 16  
3.2.1 Joining the binders .................................. 16  
3.2.2 Comparison by substitution ......................... 17  

4 Romeo ..................................................... 19  
4.1 Typing judgment ........................................ 20  
4.2 Operational semantics .................................. 21  
4.3 Execution rules ......................................... 21  
4.3.1 Quasi-literals ....................................... 24  
4.3.2 Sufficient disjointness ............................... 24  
4.4 Examples ................................................. 25  
4.4.1 Translation of `let*` ................................ 25  
4.4.2 Translation of `let` ................................... 26  

5 Freshening ................................................ 27  
5.1 Approach ............................................... 27  
5.2 Reference renaming ..................................... 27  
5.3 Implementation of freshening ............................ 28  

6 Romeo respects $\alpha$-equivalence: a guide to the proofs 30  
6.1 $\alpha$-equivalent environments yield $\alpha$-equivalent results .... 30  
6.1.1 One substitution, not one per subterm, suffices for opening ... 31  
6.1.2 Alpha-equivalence and free names are connected .......... 32  
6.2 Termination is representation-oblivious ............... 32
<table>
<thead>
<tr>
<th>Chapter</th>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>Checking binding safety statically</td>
<td></td>
</tr>
<tr>
<td></td>
<td>7.1 Odds and ends</td>
<td>36</td>
</tr>
<tr>
<td></td>
<td>7.2 Soundness of the deduction system</td>
<td>36</td>
</tr>
<tr>
<td></td>
<td>7.3 Example</td>
<td>36</td>
</tr>
<tr>
<td>8</td>
<td>Specified binding in Redex</td>
<td></td>
</tr>
<tr>
<td></td>
<td>8.1 Differences from Romeo</td>
<td>38</td>
</tr>
<tr>
<td></td>
<td>8.1.1 A change in semantics</td>
<td>39</td>
</tr>
<tr>
<td></td>
<td>8.1.2 A weakened safety guarantee</td>
<td>40</td>
</tr>
<tr>
<td></td>
<td>8.1.3 Binding specification for ellipses</td>
<td>41</td>
</tr>
<tr>
<td></td>
<td>8.2 Translation of <code>let*</code></td>
<td>42</td>
</tr>
<tr>
<td></td>
<td>8.3 Improving existing Redex models</td>
<td></td>
</tr>
<tr>
<td></td>
<td>8.3.1 Church numerals</td>
<td>42</td>
</tr>
<tr>
<td></td>
<td>8.3.2 Recursive let</td>
<td>43</td>
</tr>
<tr>
<td></td>
<td>8.3.3 π-calculus</td>
<td>43</td>
</tr>
<tr>
<td></td>
<td>8.4 Benchmarks</td>
<td>47</td>
</tr>
<tr>
<td></td>
<td>8.4.1 Microbenchmarks</td>
<td>47</td>
</tr>
<tr>
<td></td>
<td>8.4.2 Launchbury’s semantics for lazy evaluation</td>
<td>48</td>
</tr>
<tr>
<td></td>
<td>8.4.3 Church numerals</td>
<td>49</td>
</tr>
<tr>
<td></td>
<td>8.4.4 Recursive let</td>
<td>50</td>
</tr>
<tr>
<td>9</td>
<td>Related work</td>
<td>51</td>
</tr>
<tr>
<td></td>
<td>9.1 Statically specified binding in template macros</td>
<td>51</td>
</tr>
<tr>
<td></td>
<td>9.2 Pure FreshML</td>
<td>51</td>
</tr>
<tr>
<td></td>
<td>9.3 Ott</td>
<td>52</td>
</tr>
<tr>
<td></td>
<td>9.4 Hygienic macro systems in scheme</td>
<td>52</td>
</tr>
<tr>
<td></td>
<td>9.5 Binding in theorem-proving systems</td>
<td>52</td>
</tr>
<tr>
<td></td>
<td>9.6 Unbound</td>
<td>53</td>
</tr>
<tr>
<td></td>
<td>9.7 Extensions to Romeo</td>
<td>53</td>
</tr>
<tr>
<td>10</td>
<td>Future work</td>
<td>54</td>
</tr>
<tr>
<td></td>
<td>10.1 Binding features</td>
<td>54</td>
</tr>
<tr>
<td></td>
<td>10.2 Typed macros</td>
<td>55</td>
</tr>
<tr>
<td>A</td>
<td>An example derivation of $=^R$</td>
<td>57</td>
</tr>
<tr>
<td>B</td>
<td>An example derivation of $\triangleright$</td>
<td>59</td>
</tr>
<tr>
<td>C</td>
<td>Soundness of execution for $\alpha$-equivalence</td>
<td>60</td>
</tr>
<tr>
<td></td>
<td>C.1 General definitions</td>
<td>60</td>
</tr>
<tr>
<td></td>
<td>C.1.1 Naive operations</td>
<td>60</td>
</tr>
<tr>
<td></td>
<td>C.1.2 Semantics of $\beta$</td>
<td>60</td>
</tr>
<tr>
<td></td>
<td>C.2 Basic $\alpha$-equivalence lemmas</td>
<td>62</td>
</tr>
<tr>
<td></td>
<td>C.3 Execution soundness itself</td>
<td>77</td>
</tr>
<tr>
<td>D</td>
<td>Soundness of the deduction system</td>
<td>86</td>
</tr>
<tr>
<td></td>
<td>D.1 Non-occurrence of escape</td>
<td>86</td>
</tr>
<tr>
<td></td>
<td>D.2 Soundness itself</td>
<td>87</td>
</tr>
</tbody>
</table>
Chapter 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, \((\text{lambda } (x_1) e_1)\) and \((\text{lambda } (x_2) e_2)\), constructs:

\[
(\text{lambda } (x_1) (\text{lambda } (x_2) (e_1 e_2)))
\]

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

\[
(\text{lambda } (a) (\text{lambda } (b) (a b)))
\]

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

\[
(\text{lambda } (c) (\text{lambda } (c) (c c)))
\]

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 DSL implementation found that eight of them were vulnerable to errors or incorrect behavior as a result of name collision [6].

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 [11]. 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 $\lambda_m$

David Herman’s $\lambda_m$-calculus [10] is inspired by the world of Scheme macros.

A typical Scheme is a language 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 [12]. 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 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 [1]. 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` (with no Macro By Example), 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 [17] 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 nondeterministic, so Pure FreshML must ensure that the output of its programs don’t contain any of those fresh names (at least as free names; nondeterminism 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 that will never occur.

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 C\( \alpha \)ml\(^2\), which improves on Pure FreshML’s expressivity somewhat. However, even C\( \alpha \)ml’s binding specifications are significantly limited. In particular, C\( \alpha \)ml terms are divided into \textit{expressions} (which contain references) and \textit{patterns} (which contain binders), while terms in \( \lambda_m \) can play both roles simultaneously.

### 1.3 Structure of this work

We claim that it is possible to usefully manipulate syntax in an automatically binding-safe way, even syntax with complex binding patterns.

To show this, we introduce a core calculus, Romeo, which can be summarized with the slogan “Pure FreshML, but with \( \lambda_m \)’s binding specifications”. Then, to show that these ideas work in practice, we use them to add binding safety to an existing metaprogramming system, PLT Redex.

For an example of a term that a metaprogrammer might want to manipulate, consider the following Scheme term that might be data for a metaprogram:

\[
\text{(let* ((a 1)
    (b (+ a a))
    (c (* b 5)))
    (display c))}
\]

In Scheme, the \texttt{let*} syntactic form is defined to bind the names it introduces not only in the body, but also in the right hand side of each subsequent arm. Thus, in the example, all references to names are well-defined, and if it were evaluated, the value of \( c \) would be 10. This behavior is similar to the behavior of \texttt{do} in Haskell and to telescopes in dependently-typed languages.

#### 1.3.1 Romeo

Without \textit{formally} defining the above behavior, we cannot hope to reason about binding at all. Expressing binding behavior is the job of binding specifications, such as those in Nominal Isabelle\(^3\), Ott\(^4\), and C\( \alpha \)ml\(^5\).

In order to model forms like \texttt{let*}, we must allow binders to be “exported” up from subterms (and sub-subterms, etc.) in a well-defined way. In the above example, the first subterm of the \texttt{let*} exports the names \( a \), \( b \), and \( c \), in accordance with its specification. The specification of \texttt{let*} is able to treat that set of names as a unit, bringing them into scope in its second subterm. Furthermore, a name may participate in multiple binding relationships at different levels. We discuss Romeo’s binding specification system (which

---

1Pure FreshML is based on FreshML\(^6\), which stops here and achieves a less strong safety guarantee.
is built in to its type system) in chapter 2, and informally define its behavior. We show a type (with binding information) for expressions in a language that includes let* (Figures 2.1 and 2.2).

Because we need a notion of $\alpha$-equivalence, we formally define $\alpha$-equivalence over terms in chapter 3. In our system, we will be able to define binding rules such that the following terms are all $\alpha$-equivalent:

\[(\text{let* } ((a 1) (\text{let* } ((x 1) (\text{let* } ((d 1) (b (+ a a)) (y (+ x x)) (d (+ d d)) (c (* b 5))) (z (* y 5))) (d (* d 5))) (display c)) (display z) (display d))\]

In order to manipulate terms, we need a programming language. We define the core calculus Romeo with a big-step semantics in chapter 4. Romeo's destructuring construct, open, is of particular interest; it is key to the correctness of the language. (open is similar to the "val $<r_1, r_2> = e$" construct in FreshML [21] or the case construct in Pure FreshML [18]). Prior to destructuring a term, it $\alpha$-converts it so that it satisfies a property that we call 'sufficient disjointness'. For example, in an execution environment containing the name a as a free name, the middle of the above three $\alpha$-equivalent terms is the only one with sufficient disjointness.

The semantics of Romeo is underspecified in one way: although sufficient disjointness is well-defined, an algorithm for finding a sufficiently-disjoint term is needed. In chapter 5, we provide an algorithm for finding an $\alpha$-equivalent, sufficiently-disjoint term. Conjecture 8.1 states that the algorithm is correct.

To show that Romeo achieves its goal of $\alpha$-equivalence preservation, we want to show that, if any of the above $\alpha$-equivalent terms (or in general, any $\alpha$-equivalent inputs) were given as input to any Romeo program, the output of the program would be the same (up to $\alpha$-equivalence). Thus, even though Romeo programs can destruct elements and compare atoms, they cannot distinguish $\alpha$-equivalent inputs, and therefore errors of accidental name collision cannot occur. This is ensured by the theorems C.1 and C.2. Chapter 6 is a readable introduction to those proofs, and the proofs themselves take up appendix C.

Certain programming errors that would otherwise lead to violations of correctness are detected by Romeo and turned into runtime errors. For example, a program that took a let* term and attempted to return its first binder (in the examples above, a, x or d) instead returns fault. In section 7, we elaborate on Romeo's deduction system, which, with the aid of an SMT solver, can be used to statically prove that a Romeo program never returns fault. Chapter D contains the proof of soundness (theorem D.1) for Romeo's deduction system.

### 1.3.2 Redex with binding specifications

Since Romeo is a core calculus, writing and executing programs in it is not really going to make anyone’s life better. PLT Redex [7], on the other hand, is a practical tool for defining languages and their semantics. Prior to our work, it lacked any binding-specific tools. We have augmented it with Romeo-inspired binding specifications and a corresponding change to the way it destructures terms. We omit one part of Romeo: the detection of name escape and generation of faults. We conjecture a weakened safety guarantee (conjecture 8.1) and further claim that it is almost as good as full $\alpha$-equivalence preservation.

Redex lacks a type system entirely, so the binding behavior of terms is provided by the structure of the values themselves. This is consistent with Redex's general philosophy. We also introduce features to make binding structures more ergonomic and useful.

To demonstrate the usefulness of our changes to Redex, we take some pre-existing Redex models and add binding forms to them. At the cost of adding three or four lines of specification, we remove many lines of name-handling code, or, in one case, fix a known
bug in name safety. We also examine the performance cost of our changes. The performance cost is significant, but decreased compile time from code size reduction can outweigh that and improve the user experience.
Chapter 2

Binding language

2.1 Overview of binding types

Values in our system are plain old data, that is, S-expressions or something similar. We use binding types to specify the binding properties of these terms.

Binding types augment a traditional context-free grammar with a single attribute (in the style of an attribute grammar) that represents the flow of bindings from one subterm to another. In Figure 2.1, the type definition of \texttt{CoreExpr} looks like a traditional grammar for the lambda calculus, with one major difference: the notation \texttt{CoreExpr↓0} indicates that the binder “exported” by the child in position 0 (an atom with type \texttt{BAtom} exports itself) is to be in scope in the \texttt{CoreExpr} body of the lambda. \footnote{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. It is possible to imagine a system in which old names are removable (e.g., a construct \texttt{(unbind x e)}, in which the name \texttt{x} is not a valid reference in \texttt{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 \cite{9}.)} The names exported by a product value are defined by export notation on the product type, marked with ↑.

To facilitate the matching of related names together, the type for name introductions (or binders), \texttt{BAtom}, is made distinct from the type of names that reference binders, \texttt{RAtom}.

Instead of a binary product (×), our type system’s product combines an arbitrary number of types (\texttt{Prod}↑β↑ε (τ↓β↓i)). Using a “wide” product increases the expressivity of our system because the position references on the right-hand-side of ↓ (and ↑) 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 defines bindings in products by first identifying what names are exported by each child (e.g. a \texttt{BAtom} or a \texttt{Prod} with a non-empty ↑), and then determining which names are imported by which children. The latter is the responsibility of the ↓ operator. Considering the more complex example 2.2, consider \texttt{let*}, in \texttt{Expr}. The “sequential let” line indicates that the binders exported by \texttt{LetStarClauses} are in scope in the body of the \texttt{let*} expression. The grammar for \texttt{LetStarClauses} says that a set of \texttt{let*}-clauses is either the

\begin{verbatim}
CoreExpr ::= RAtom            (variable reference)
    | Prod (CoreExpr, CoreExpr)  (application)
    | Prod (BAtom, CoreExpr↓0)   (abstraction)
\end{verbatim}

\footnote{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. It is possible to imagine a system in which old names are removable (e.g., a construct \texttt{(unbind x e)}, in which the name \texttt{x} is not a valid reference in \texttt{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 \cite{9}.)}

Figure 2.1: Example type for the lambda calculus.
empty list or the cons of a single clause and a LetStarClauses. In the latter case, there are two clauses:

- Prod⁰⁰ (BAtom, Expr) is itself made of two clauses:
  - the BAtom is exported from that clause, as indicated by ⇑0,
  - the Expr is left alone (in particular, the BAtom is not in scope for it),
- the other clause, LetStarClauses↓0 indicates that the names (in this case, just one) exported from the clause in position 0 is in scope for the LetStarClauses,
- and ⇑1▷0 indicates that both clauses are exported. The notation ▷ indicates that names from the clause in position 1 (the LetStarClauses) take priority over names from the clause in position 0, if any are the same.

If we had wanted to specify that all the binders in a let* must be distinct, then we could have written ⇑(1⊎0), which behaves like ⇑(1▷0), except that constructing a value with duplicated binders is 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 ↓, ⇑, ▷, and ⊎ 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; they are described in more detail in section 2.3.

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 2.3, 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:

\[
\begin{aligned}
&\text{(handler a} \\
&\quad \text{b (deal-with a b)} \\
&\quad \text{b (put-tag a (text-of b)))}
\end{aligned}
\]

The scope of the first \(b\) is the \(\text{(deal-with} \ldots)\), and the scope of the second one is the \(\text{(put-tag} \ldots)\). Regardless of whether they have the same names, the meanings of the two events must not be conflated, but in both subterms, gui-elt is the same.

Our goal of supporting realistic concrete syntax is particularly relevant here. The user could have implemented the handler statement as a \textit{function} with the following style of expected invocation:

\[
\begin{aligned}
&\text{(handler-fn (lambda (gui-elt mouse-evt)} \\
&\quad \text{(deal-with gui-elt mouse-evt))} \\
&\quad \text{(lambda (gui-elt kbd-evt)} \\
&\quad \text{(put-tag gui-elt (text-of kbd-evt))})
\end{aligned}
\]

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.

\[
\begin{aligned}
\tau \in \text{Type} &::= \text{BAtom} \\
&\mid \text{RAAtom} \\
&\mid \tau + \tau \\
&\mid \text{Prod}^{\beta} (\tau_0 \downarrow \beta_0, \ldots, \tau_n \downarrow \beta_n) \\
&\mid \mu X. \tau \\
&\mid X \\
&\mid \text{nth}_j \tau
\end{aligned}
\]

Values are either atoms, left- or right- injections of values (to model sum types), or tuples of values. We write \text{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 \text{BAtom} (for binders) and \text{RAAtom} (for references). These types tell us how to interpret atoms. By convention, \text{BAtoms} export themselves and \text{RAAtoms} export nothing.

Tuples are interpreted by \text{Prod} types. The type \text{Prod}^{\beta\alpha} \((\tau_0 \downarrow \beta_0, \ldots, \tau_n \downarrow \beta_n)\), which we denote by the comprehension \text{Prod}^{\beta\alpha} \((\tau_i \downarrow \beta_i)_i\), tells us how to interpret the value \text{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 \(v_i\). The binding combinator \(\beta_\alpha\), 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 \text{prod} \((v_i)_i\). If \(\downarrow\) or \(\uparrow\) is omitted, the corresponding \(\beta\) defaults to \(\emptyset\).

To avoid complicating our core calculus with issues of parsing, we have explicit sum types and injections. A value \text{inj}0 \((v)\) (resp. \text{inj}1 \((v)\)) is interpreted by the type \(\tau_0 + \tau_1\) so that \(v\) is interpreted by \(\tau_0\) (resp. \(\tau_1\)). A more realistic system would use a parser (see chapter 8).

To support recursive types, we have type variables \(X\), and type-level destructors \text{nth}_j. We define the latter as \text{nth}_j \text{Prod}^{\beta\alpha} \((\tau_i \downarrow \beta_i)_i \triangleq \tau_j\) (reusing our existing type constructor as a way to write type
tuples; the $\beta_s$ and $\betaex_s$ 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:

$$
\ell \in \mathbb{N} \\
\beta \in \text{Beta} ::= \emptyset | \beta \cup \beta | \beta \triangleright \beta | \ell
$$

We can determine whether a particular index is referred to by $\beta$:

$$
\_ \in \_ \subseteq \mathbb{N} \times \text{Beta} \\
\ell \in \emptyset \triangleq \text{false} \\
\ell \in \ell' \triangleq (\ell = \ell') \\
\ell \in \beta \triangleright \beta' \triangleq \ell \in \beta \text{ or } \ell \in \beta' \\
\ell \in \beta \cup \beta' \triangleq \ell \in \beta \text{ or } \ell \in \beta'
$$

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

$$
[\_] (\_) : \text{Beta} \times \text{AtomSet}^{*} \rightarrow \text{AtomSet} \\
\langle \emptyset \rangle (A_i)_{i} \triangleq \emptyset \\
\langle \ell \rangle (A_i)_{i} \triangleq A_{\ell} \\
\langle \beta \triangleright \beta' \rangle (A_i)_{i} \triangleq [\beta] (A_i)_{i} \cup [\beta'] (A_i)_{i} \\
\langle \beta \cup \beta' \rangle (A_i)_{i} \triangleq [\beta] (A_i)_{i} \cup [\beta'] (A_i)_{i}
$$

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

In Romeo, constructing a value whose type contains a $\beta$ that contains a $\cup$ 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. Basic operations on substitutions are straightforward:

$$
\sigma(a) := a', \text{ where } \langle a, a' \rangle \in \sigma \\
\sigma(a) := a, \text{ otherwise}
$$

$$
\text{dom}(\sigma) ::= \{ a_d \mid \langle a_d, a_r \rangle \in \sigma \} \\
\text{rng}(\sigma) ::= \{ a_r \mid \langle a_d, a_r \rangle \in \sigma \}
$$

We interpret $\beta$‘s on substitutions as follows:

$$
[\_] (\_) : \text{Beta} \times \text{Subst}^{*} \rightarrow \text{Subst} \\
\langle \emptyset \rangle (\sigma_i)_{i} \triangleq \emptyset \\
\langle \ell \rangle (\sigma_i)_{i} \triangleq \sigma_{\ell} \\
\langle \beta \triangleright \beta' \rangle (\sigma_i)_{i} \triangleq [\beta] (\sigma_i)_{i} \triangleright [\beta'] (\sigma_i)_{i} \\
\langle \beta \cup \beta' \rangle (\sigma_i)_{i} \triangleq [\beta] (\sigma_i)_{i} \cup [\beta'] (\sigma_i)_{i}
$$

We define $\sigma$ “overrides” $\sigma'$ as follows:

$$
\sigma \triangleright \sigma' \triangleq \sigma \cup \left\{ \langle a_d, a_r \rangle \mid \langle a_d, a_r \rangle \in \sigma' \right\} \\
\sigma \not\triangleright \sigma' \triangleq \sigma \cup \left\{ \langle a_d, a_r \rangle \mid a_d \not\in \text{dom}(\sigma) \right\}
$$
The operation for combining disjoint substitutions $\sigma \sqcup \sigma'$ is like $\triangleright$, except that it is undefined if the domains of the substitutions in question overlap.

Now, using $\llbracket \beta \rrbracket (A_i)_i$, we can compute the free binders of any value, $fb(\tau, v)$. This is equal to the set of names that is exported from it.

$$fb(\text{Prod}^{\triangleright} (\tau_i \downarrow \beta_i)_i \cdot \text{prod} (v_i)_i) \triangleq \llbracket \beta_\alpha \rrbracket (fb(\tau_i, v_i))_i$$
$$fb(\tau_0 + \tau_1, \text{inj}_0(v)) \triangleq fb(\tau_0, v)$$
$$fb(\tau_0 + \tau_1, \text{inj}_1(v)) \triangleq fb(\tau_1, v)$$
$$fb(\mu X.\tau, v) \triangleq fb(\tau[\mu X.\tau/X], v)$$
$$fb(\text{BAtom}, a) \triangleq \{a\}$$
$$fb(\text{RAtom}, a) \triangleq \emptyset$$

There are several other useful quantities that we can compute using these combiners. First is the set of free references of a term. The product case is again the interesting case, with set subtraction occurring to remove references that are bound by the product itself.

$$fr(\text{Prod}^{\triangleright} (\tau_i \downarrow \beta_i)_i \cdot \text{prod} (v_i)_i) \triangleq \bigcup_i (fr(\tau_i, v_i) \setminus \llbracket \beta_i \rrbracket (fb(\tau_j, v_j))_j)$$
$$fr(\tau_0 + \tau_1, \text{inj}_0(v)) \triangleq fr(\tau_0, v)$$
$$fr(\tau_0 + \tau_1, \text{inj}_1(v)) \triangleq fr(\tau_1, v)$$
$$fr(\mu X.\tau, v) \triangleq fr(\tau[\mu X.\tau/X], v)$$
$$fr(\text{BAtom}, a) \triangleq \emptyset$$
$$fr(\text{RAtom}, a) \triangleq \{a\}$$

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

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

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:

$$xa(\text{Prod}^{\triangleright} (\tau_i \downarrow \beta_i)_i \cdot \text{prod} (v_i)_i) \triangleq \left( \bigcup_i fb(\tau_i, v_i) \right) \setminus fb(\text{Prod}^{\triangleright} (\tau_j \downarrow \beta_j)_j \cdot \text{prod} (v_j)_j)$$
Chapter 3

\(\alpha\)-equivalence

Our next task is to go from a binding type to a notion of \(\alpha\)-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 \(\alpha\)-equivalent if

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

\[\_ =_{\alpha} \_ \subseteq \text{Value} \times \text{Value} \times \text{Type}\]

\[
\frac{v =_B v' : \tau}{\frac{v =_R v' : \tau}{v =_{\alpha} v' : \tau}} \quad \text{\(\alpha\)EQ}
\]

We use \(=_B\) (pronounced “binder-equivalent”) for the first relation and \(=_R\) (pronounced “reference-equivalent”) for the second. The calculation of \(=_R\) is similar to the conventional notion of \(\alpha\)-equivalence.

3.1 Binder equivalence

Two values are \(=_B\) iff their exported (free) binders in the same positions are identical. Note that non-exported binders are irrelevant to \(=_B\), but are important to the calculation of \(=_R\).

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

\[\_ =_B \_ \subseteq \text{Value} \times \text{Value} \times \text{Type}\]

\[
\frac{a =_B a : \text{BAtom}}{\text{BAtom-BAtom}} \quad \frac{a =_B a' : \text{RAtom}}{\text{BAtom-RAtom}} \quad \frac{\forall i \in \beta_{\text{ex}}. v_i =_B v'_i : \tau_i}{\text{prod}(v_i)_i =_B \text{prod}(v'_i)_i : \text{Prod}^{\beta_{\text{ex}}}(\tau_i|_{\beta_i})_i} \quad \text{BAtom-Prod}
\]

For example:

\[
\text{prod}(a \ b \ c) =_B \text{prod}(a \ b \ d) : \text{Prod}^{0_{\text{ex}}}(\text{BAtom, BAtom, BAtom})
\]
because non-exported atoms are ignored, but:
\[
\prod (a \ b \ c) \neq_B \prod (b \ a \ c) : \text{Prod}^{0b_1} (\text{BAtom}, \text{BAtom}, \text{BAtom})
\]
because exported names in each position must be the same.

### 3.2 Reference equivalence

Calculating \(=_R\) is analogous to the conventional notion of \(\alpha\)-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 \(\bowtie\ \frown\) 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 \(\bowtie\ \frown\) 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 \(J^\upbeta_{\text{ex}}\). The resulting pair of substitutions is the output of the \(\bowtie\ \frown\) relation.

\[
\bowtie\ \frown:\ Value \times Value \times Type \times Subst \times Subst \subseteq Value \times Value \times Type \times Subst \times Subst
\]

\[
\frac{a \bowtie a' : \text{BAtom} \rightarrow \{\langle a, a_{\text{fresh}} \rangle\} \bowtie \{\langle a', a_{\text{fresh}} \rangle\}}{a \bowtie a' : \text{BAtom} \rightarrow \emptyset \bowtie \emptyset}
\]

\[
\frac{\forall i. v_i \bowtie v'_i : \tau_i \rightarrow \sigma_i \bowtie \sigma'_i \quad \sigma = [\beta_{\text{ex}}] (\sigma_i) \quad \sigma' = [\beta_{\text{ex}}] (\sigma'_i)}{\prod (v_i) \bowtie \prod (v'_i) : \text{Prod}^{[\beta_{\text{ex}}]} (\tau_i \downarrow \beta_i) \rightarrow \sigma \bowtie \sigma'}
\]

Here and elsewhere, we use \# to denote the disjointness operator over names, sets of names, and (the names in) values. When it examines values, it does so naively, ignoring binding structure.

For example, consider the two \texttt{let*} expressions we have previously discussed:

\[
\begin{align*}
\text{(let* ((a 1)) (b (+ a a)) (c (* b 5)))} \quad & \quad \text{(let* ((d 1)) (d (+ d d)) (display c))}
\end{align*}
\]

The results of \(\bowtie\) on their subterms in position 1 (the \texttt{display} expressions) are \(\emptyset\) and \(\emptyset\), because neither one has any free binders. Position 0 corresponds to the \texttt{LetStarClauses}, and is more interesting. \(\bowtie\) will nondeterministically generate three names, which we will choose to be aa, bb, and cc. Then, we will have \(\sigma_0 = \{\langle a, aa \rangle, \langle b, bb \rangle, \langle c, cc \rangle\}\) and \(\sigma'_0 = \{\langle d, cc \rangle\}\). 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 can be found in appendix B.
3.2.2 Comparison by substitution

Now we can write the rules for \(\equiv R\). At the type RAAtom, the atoms being compared are necessarily free, and are reference equal iff they are identical. Symmetrically to \(\equiv B\), any two atoms are \(\equiv R\) at BAtom.

At a wide product, the information from performing \(\infty\) 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 \(\infty\) 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 \(\llbracket \beta \rrbracket (\sigma_j_i)(v_i)\) (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 \(\infty\)).

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 naivety is acceptable because, first, \(\equiv R\) does not examine free binders, and second, (broadly speaking) the substitution of un-free names is harmless (see Lemma C.31). Because our substitutions are naive, we require that each substitution’s range be disjoint from the values being examined (recall that \(\infty\) nondeterministically chooses the substitutions’ ranges). Without this requirement, we would have \((\text{let}^* ((x \ 7)) \ x) \equiv R (\text{let}^* ((y \ 7)) \ a)\): Expr, witnessed by \(\sigma_0 = \{\langle x, a \rangle\}\) and \(\sigma'_0 = \{\langle y, a \rangle\}\). This description is captured by the following rules:

\[
\begin{align*}
- \equiv R \quad &- - - \subseteq \text{Value} \times \text{Value} \times \text{Type} \\
\forall i, v_i \equiv v'_i : \tau_i \rightarrow \sigma_i \equiv \sigma'_i \quad &\forall i, j, \text{rng}(\sigma_i) \neq \text{rng}(\sigma'_j) \quad &\forall i, \llbracket \beta_i \rrbracket (\sigma_i_j_i)(v_i) \equiv R \llbracket \beta'_i \rrbracket (\sigma'_j_i)(v'_i) : \tau_i \\
\text{prod}(v_i) \equiv R \text{prod}(v'_i) : \text{Prod}^{\beta_\text{in}}(\tau_i \downarrow \beta_i) \\
\end{align*}
\]

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

An example that better demonstrates the complexities of renaming is the event handler example from section 2.1.1. The result of invoking \(\infty\) on each pair of children, in order to compare the two versions for \(\alpha\)-equivalence, is in Figure 3.1.

Because the corresponding subterms import nothing, \(\beta_0 = \beta_1 = \beta_3 = \emptyset, \beta_2\) is \(0 \uplus 1\) and \(\beta_4\) is \(0 \uplus 3\). The result of performing those substitutions is shown in Figure 3.2, establishing the relationship between mouse-evt and the first b and the relationship between kbd-evt and the second b.
<table>
<thead>
<tr>
<th>( j )</th>
<th>( v_j )</th>
<th>( v'_j )</th>
<th>( \sigma_j )</th>
<th>( \sigma'_j )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>gui-elt ( \bowtie ) a</td>
<td>: BAtom ( \rightarrow ) {gui-elt, gg} ( \bowtie ) {a, gg}</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>mouse-evt ( \bowtie ) b</td>
<td>: BAtom ( \rightarrow ) {mouse-evt, mm} ( \bowtie ) {b, mm}</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>(handle-evt ...) ( \bowtie ) (handle-evt ...)</td>
<td>: Expr ( \rightarrow ) \emptyset ( \bowtie ) \emptyset</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>kbd-evt ( \bowtie ) b</td>
<td>: BAtom ( \rightarrow ) {kbd-evt, kk} ( \bowtie ) {b, kk}</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>(tag ...) ( \bowtie ) (tag ...)</td>
<td>: Expr ( \rightarrow ) \emptyset ( \bowtie ) \emptyset</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 3.1: Substitutions generated for the handler example

\[
\begin{align*}
0 \cup 1 &\langle (\text{handle-evt gui-elt mouse-evt}) \rangle = (\text{handle-evt gg mm}) \\
0 \cup 1 &\langle (\text{handle-evt a b}) \rangle = (\text{handle-evt gg mm}) \\
0 \cup 3 &\langle (\text{put-tag gui-elt (text-of kbd-evt)}) \rangle = (\text{put-tag gg (text-of kk)}) \\
0 \cup 3 &\langle (\text{put-tag a (text-of b)}) \rangle = (\text{put-tag gg (text-of kk)})
\end{align*}
\]

Figure 3.2: Result of substitution in the handler example (all other subterms are trivial)
Chapter 4

Romeo

Romeo\(^1\) is a first-order, typed, side-effect-free language with the values and types of section 2.2. It uses the types to direct the interpretation of these values 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 \textit{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 7) to generate proof obligations which, if satisfied, guarantee that escape (and thus, \textit{FAULT}) will never occur.

The syntax of Romeo is given as follows:

\[
\begin{align*}
p &\in \text{Prog} ::= fD . . . e : \tau \\
fD &\in \text{FnDef} ::= (\text{define-fn} \ f x : \tau \ldots) : \tau \ e \\
e &\in \text{Expr} ::= (f x \ldots) \\
&\quad | \ (\text{fresh} \ x \in \ e) \\
&\quad | \ (\text{let} \ x \ be \ e \ in \ e) \\
&\quad | \ (\text{case} \ x \ (x \ e) \ (x \ e)) \\
&\quad | \ (\text{open} \ x \ (x \ldots) \ e) \\
&\quad | \ (\text{if} \ x \ equals \ x \ e \ e) \\
\end{align*}
\]

\(e_{\text{qlit}} \in \text{QuasiLit} ::= x\)

- \(\text{ref} \ x\)
- \(\text{inj}_0 \ e_{\text{qlit}} \tau\)
- \(\text{inj}_1 \tau \ e_{\text{qlit}}\)
- \(\text{prod}^{\phi \beta} (e_{\text{qlit}}^{i \downarrow \beta_i})\)

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 \textit{typeof}(\Gamma, e) whose definition is routine.

\(^1\)As noted nominal logician Juliet Capulet observed, Romeo’s name is irrelevant.
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 \(\text{open}\)). As a result, programs are written in monadic form (similar to A-normal form), naming intermediate results with \(\text{let}\).

### 4.1 Typing judgment

Although Romeo’s types are unusual, the type system itself is straightforward. This is because the binding annotations do not rule out any operations, but rather change the behavior of \(\text{open}\) and \(\text{fresh}\).

\[
\begin{align*}
  \text{T-FRESH} & \quad \frac{\text{x fresh for } \Gamma}{\Gamma \vdash \text{fresh } \text{x in } e : \tau} \\
  \text{T-OPEN} & \quad \frac{\Gamma(\text{x}_{\text{obj}}) = \text{Prod}^{\beta_\text{x}} (\tau_{\text{i}} \downarrow \beta_{\text{i}})_{\text{i}} \quad \forall i. \text{x}_{\text{i}} \text{ fresh for } \Gamma}{\Gamma \vdash \text{open } x_{\text{obj}} ((x_{\text{i}})_{\text{i}}) e : \tau}
\end{align*}
\]

\[
\begin{align*}
\text{T-CALL} & \quad \frac{\text{argtype}(f) = (\tau)_{\text{i}} \quad \text{rettype}(f) = \tau}{\forall i. \Gamma(\text{x}_{\text{i}}) = \tau_{\text{i}}}
\end{align*}
\]

\[
\begin{align*}
\text{T-LET} & \quad \frac{x \text{ fresh for } \Gamma}{\Gamma \vdash \text{let } C \text{ be } e_{\text{x}} \text{ in } e_{\text{body}} : \tau_{\text{body}}} \\
\text{T-IFEQ} & \quad \frac{\forall i. \Gamma(\text{x}_{\text{i}}) = \tau_{\text{i}}}{\Gamma \vdash \text{if } x_{\text{0}} \text{ equals } x_{\text{1}} e_{\text{0}} e_{\text{1}} : \tau}
\end{align*}
\]

\[
\begin{align*}
\text{T-PROD} & \quad \frac{\Gamma \vdash \text{prod}^{\beta_{\text{x}}} (e_{\text{i}} \downarrow \beta_{\text{i}})_{\text{i}}}{\Gamma \vdash \text{prod}^{\beta_{\text{x}}} (\tau_{\text{i}} \downarrow \beta_{\text{i}})_{\text{i}}} \\
\text{T-INJ0} & \quad \frac{\Gamma \vdash \text{inj}^{0} : \tau_{\text{left}} \downarrow \tau_{\text{right}}}{\Gamma \vdash \text{inj}^{0} : \tau_{\text{left}} + \tau_{\text{right}}}
\end{align*}
\]

\[
\begin{align*}
\text{T-QLIT-INJ1} & \quad \frac{\Gamma \vdash \text{inj}^{1} : \tau_{\text{left}} \downarrow \tau_{\text{right}}}{\Gamma \vdash \text{inj}^{1} : \tau_{\text{left}} + \tau_{\text{right}}} \\
\text{T-QLIT-VAR} & \quad \frac{\Gamma(\text{x}) = \tau}{\Gamma \vdash \text{x} : \tau}
\end{align*}
\]

\[
\begin{align*}
\text{T-QLIT-REF} & \quad \frac{\Gamma(\text{x}) = \text{BAtom}}{\Gamma \vdash \text{ref } x : \text{RAtom}} \\
\text{T-FND} & \quad \frac{\forall i. \text{x}_{\text{i}} : \tau_{\text{i}} \vdash \epsilon \text{ : } \tau_{0}}{\Gamma \vdash \text{define-fn } (f (x_{\text{i}} : \tau_{\text{i}})_{\text{i}} \text{ pre } C_{\text{pre}}) : \tau_{\text{i}} \epsilon \text{ post } C_{\text{post}}}
\end{align*}
\]

\[
\begin{align*}
\text{T-PROG} & \quad \frac{\forall i. \epsilon \vdash f D_{\text{i}} : \tau}{\Gamma \vdash \epsilon : \tau}
\end{align*}
\]
4.2 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 the names generated by the fresh and open forms.

We begin with some auxiliary definitions that we will need:

\[
\begin{align*}
\text{Result} & \supseteq v | \text{FAULT} \\
\text{ValEnv} & \mathrel{::=} \epsilon | \rho[x \mapsto v] \\
\text{TypeEnv} & \mathrel{::=} \epsilon | \Gamma, x : \tau \\
fa_{\text{env}}(\Gamma, \rho) & \mathrel{=} \bigcup_{x \in \text{dom}(\Gamma)} fa(\Gamma(x), \rho(x))
\end{align*}
\]

The form of the execution judgment is:

\[
\Gamma \vdash_{\text{exe}} (\epsilon, \rho) \Downarrow w
\]

The \(k\) argument indicates the number of execution steps taken to produce the result \(w\). It exists to proof by induction on execution rules possible.

4.3 Execution rules

We can now give the rules for execution in Romeo. Rules that introduce names come in two forms, -Ox, 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.

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 C.1
and C.2) guarantee that the choice of the new name will not affect the result (up to α-equivalence). We have two versions of the rule: Fresh-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 quasiliteral (ref x) (see section 4.3.1).

The hypothesis τ = typeof ((Γ, x:BAtom), e) 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.

\[
\frac{a \notin \text{fa}_\text{env} (\Gamma, \rho) \quad \Gamma, x: \text{BAtom} \vdash e, \rho [x \rightarrow a]}{\text{E-Fresh-Fail}}
\]

\[
\frac{\tau = \text{typeof} ((\Gamma, x: \text{BAtom}), e) \quad w = \text{FAULT} \lor a \notin \text{fa}(\tau, w)}{\Gamma \vdash e, \rho [x \rightarrow a]} \quad \text{E-Fresh-Ok}
\]

The next pair of rules safely destructures a product value. When given the input \(\rho(x_{obj}) = \prod_{i=1}^{n} (v_{obj,i})\), the open expression chooses an \(\alpha\)-variant \(\prod_{i=1}^{n} (v_{1}, \ldots, v_{n})\) 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 4.3.2. 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 exposable atoms of the renamed input. The fact that only those names must be renamed to achieve safety is one of the consequences of Lemma 6.1.1, discussed in section 6.1.1.

\[
\frac{\rho(x_{obj}) = \alpha \prod (v_{1}) \quad \tau_{obj} = \Gamma (x_{obj}) \quad \text{fa} (\Gamma, \rho) \vdash_{\text{suff-disj}} \prod (v_{1}) \quad \Gamma, (x_{i} : \tau_{i}) \vdash e, \rho [x_{i} \rightarrow v_{1(i)}]}{\text{E-Open-Ok}}
\]

\[
\frac{\tau = \text{typeof} (\Gamma, e) \quad w = \text{FAULT} \lor \text{fa}(\tau, w)}{\Gamma \vdash e, \rho [x_{i} \rightarrow v_{1(i)}]} \quad \text{E-Open-Fail}
\]

There are two evaluation rules for let, depending on whether calculating \(e_{\text{val}}\) faults. As noted above, the constraint \(C\) is ignored at run-time.
\[ \Gamma \vdash \text{exe} \langle e_{\text{val}}, \rho \rangle \xrightarrow{k_{\text{val}}} v_{\text{val}} \]

\[ \tau_{\text{val}} = \text{typeof} (\Gamma, e_{\text{val}}) \quad \Gamma, x : \tau_{\text{val}} \vdash \text{exe} \langle e_{\text{body}}, \rho [x \rightarrow v_{\text{val}}] \rangle \xrightarrow{k_{\text{body}}} w \quad \text{E-LET} \]

\[ \Gamma \vdash \text{exe} \langle \{\text{let } x \text{ where } C \text{ be } e_{\text{val}} \text{ in } e_{\text{body}} \}, \rho \rangle \xrightarrow{k_{+1}} w \quad \text{E-LET-FAIL} \]

As in Pure FreshML [18], 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.

\[
\begin{aligned}
\text{body}(f) &= e \\
\text{formals}(f) &= (x_{\text{formal},i} : \tau_{\text{formal},i})_i \\
\langle x_{\text{formal},i} : \tau_{\text{formal},i} \rangle_i \vdash \text{exe} \langle e_i [x_{\text{formal},i} \rightarrow \rho_i] \rangle_{\text{formal},i}) \xrightarrow{k} w
\end{aligned}
\]

\[ \Gamma \vdash \text{exe} \langle \{\text{let } f \text{ be } e_{\text{actual},i} \}, \rho \rangle \xrightarrow{k_{+1}} w \quad \text{E-CALL} \]

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.

\[ \rho(x_1) = a \quad \rho(x_r) = b \quad a = b \quad \Gamma \vdash \text{exe} \langle e_{0}, \rho \rangle \xrightarrow{k} w \quad \text{E-IF-YES} \]

\[ \rho(x_1) = a \quad \rho(x_r) = b \quad a \neq b \quad \Gamma \vdash \text{exe} \langle e_{1}, \rho \rangle \xrightarrow{k} w \quad \text{E-IF-NO} \]

\[ \rho(x_{\text{obj}}) = \text{inj0}(v_0) \quad \Gamma(x_{\text{obj}}) = \tau_0 + \tau_1 \quad \Gamma, x_0 : \tau_0 \vdash \text{exe} \langle e_0, \rho [x_0 \rightarrow v_0] \rangle \xrightarrow{k} w \quad \text{E-CASE-LEFT} \]

\[ \rho(x_{\text{obj}}) = \text{inj1}(v_1) \quad \Gamma(x_{\text{obj}}) = \tau_0 + \tau_1 \quad \Gamma, x_1 : \tau_1 \vdash \text{exe} \langle e_1, \rho [x_1 \rightarrow v_1] \rangle \xrightarrow{k} w \quad \text{E-CASE-RIGHT} \]

The E-Proc 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.

\[ \epsilon \vdash \text{exe} \langle e, \epsilon \rangle \xrightarrow{k} v \quad \Gamma \vdash \text{exe} f \ldots e \xrightarrow{k_{+1}} v \quad \text{E-PROC} \]
4.3.1 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.

\[
\begin{align*}
  v &= \text{qe} \text{lit}\_\rho \\
  \Gamma \vdash_{\text{exe}} \langle \text{qe} \text{lit}, \rho \rangle &\Rightarrow v
\end{align*}
\]

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

\[
\begin{align*}
  \text{J} \text{\_} K &\colon \text{QuasiLit} \times \text{ValEnv} \to \text{Value} \\
  \text{J} x K &\equiv \rho (x) \\
  \text{J} (\text{ref} x) K &\equiv \text{inj} 0 (\text{qe} \text{lit} \_\rho) \\
  \text{J} (\text{inj} 0 \tau e \text{qlit}) K &\equiv \text{inj} 1 (\text{qe} \text{lit} \_\rho) \\
  \text{J} (\text{prod} \_\beta \_\text{ex} (e \text{qlit}_i \downarrow \beta_i)) K &\equiv \text{prod} (\text{qe} \text{lit}_i \_\rho)_i
\end{align*}
\]

4.3.2 Sufficient disjointness

The requirement that evaluation be insensitive to \(\alpha\)-equivalent inputs leads to strong requirements on the way that open destructures values. Consider the \texttt{let*} example from before:

\[
\begin{align*}
\text{(let* ((a 1) (let* ((d 1) (b (+ a a)) (d (+ d d)) (c (* b 5))) (d (* d 5))) (display c)) (display d))
\end{align*}
\]

These are \(\alpha\)-equivalent, but if they were each destructured without renaming, we would have \(d = d = d\), even though \(a \neq b \neq c\), violating our goal of being indifferent to \(\alpha\)-conversion. Therefore, E-Open needs to freshen each binder to a distinct new name, e.g.

\[
\begin{align*}
\text{(let ((aa 1) (bb (+ aa aa)) (cc (* bb 5))) (display cc))}
\end{align*}
\]

The rule to ensure this is that, before destructuring, we must \(\alpha\)-convert values so that the binders exposed by destructuring are disjoint from each other and from any names that appear in the environment. This gives rise to the hypothesis

\[
\begin{align*}
  \vdash_{\text{suff-disj}} \text{prod} (v_i)_i : \tau_{\text{obj}}
\end{align*}
\]

in the E-Open* 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.

\[
\begin{align*}
  \vdash_{\text{bndrs-disj}} a : \text{BAtom} &\quad \text{BD-BATOM} \\
  \vdash_{\text{bndrs-disj}} a : \text{RAtom} &\quad \text{BD-RATOM}
\end{align*}
\]

\[
\begin{align*}
  \forall i, j \in \beta_{\text{ex}}. i \neq j &\Rightarrow \text{fb} (\tau_i, v_i) \neq \text{fb} (\tau_j, v_j) \\
  \forall i \in \beta_{\text{ex}}. \vdash_{\text{bndrs-disj}} v_i : \tau_i &\Rightarrow \vdash_{\text{bndrs-disj}} \text{prod} (v_i)_i : \text{Prod}^{\#\beta_{\text{ex}}} (\tau_i, \beta_i)_i
\end{align*}
\]

BD-Prod
We can now define \( \vdash \) 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 \( \text{xa} \)) are disjoint from a set \( A \) of atoms (in practice, this is the set of free atoms in the environment).

\[
\forall i \notin \beta_{\text{ex}}. \vdash_{\text{bndrs-disj}} v_i : \tau_i \quad \forall i, j \notin \beta_{\text{ex}}, i \neq j \Rightarrow \text{fb}(\tau_i, v_i) \# \text{fb}(\tau_j, v_j) \\
\text{xa}(\text{prod}(v_i)_i, \text{Prod}^{\beta_{\text{ex}}}(\tau_i_{\downarrow \beta_i})_i) \# A \\
A \vdash \text{suff-disj} \text{prod}(v_i)_i : \text{Prod}^{\beta_{\text{ex}}}(\tau_i_{\downarrow \beta_i})_i
\]

### 4.4 Examples

#### 4.4.1 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 4.2, mentions types defined in Figure 4.1 (the same as the types described in Figures 2.1 and 2.2). It is written in Romeo-L [14], which is a friendlier frontend 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 7.3) 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 4 ways depending on whether the \( \text{Expr} \) 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 \( \text{Expr} \) 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).
Figure 4.2: A Romeo-L function to expand away `let*`.

Lines 6–11 destruct the `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 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`.

### 4.4.2 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:

\[
\text{LetClauses ::= Prod } \mid \text{Prod}^{\downarrow 0} (\text{Prod}^{\downarrow 0} (\text{BAtom}, \text{Expr}), \text{LetClauses})
\]

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 4.2 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 C.1, which ensures that programs cannot observe anything about names except their binding structure, as defined by their binding specifications.
Chapter 5

Freshening

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

5.1 Approach

Conceptually, the \( \vdash_{\text{suff-disj}} \) predicate (motivated and defined in section 4.3.2) 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:

\[
(\text{let*} \ ((x (\text{string-length } y))
(\text{y } 5)
(\text{z (+ y 1)}))
(+ y 2))
\]

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

Therefore, after generating a fresh replacement for \( y \), we must use \( \downarrow \) (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.
\[ \sigma|_\tau (\cdot|\cdot) : \text{Subst} \times \text{Value} \times \text{Type} \rightarrow \text{Value} \]
\[ \sigma|_\tau (a : \text{BAtom}) \triangleq a \]
\[ \sigma|_\tau (a : \text{RAtom}) \triangleq \sigma(a) \]
\[ \sigma|_\tau (\text{prod}(v_i)_i : \text{Prod}^{\beta_a \upharpoonright} (\tau_i \downarrow \beta_i)_j) \triangleq \text{prod}(\sigma|_\tau (v_i)_i : \tau_i)_j \]
where \( \forall i. \ A_i \triangleq \llbracket \beta_i \rrbracket (\text{fb}(v_j, \tau_j))_j \)

5.3 Implementation of freshening

Let \( \nu_{\text{top}} \) denote the term we are freshening, and \( \nu'_{\text{top}} \) denote the result of freshening.

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

The function freshen-subterm applied to \( \nu \) returns a value \( \nu' \) and a substitution \( \sigma \). We conjecture that the following crucial properties hold:

- \( A \vdash \text{suff-disj } \nu' : \tau \) (this is the crucial\(^1\) property, specified in section 4.3.2),
- \( \nu =_R \nu' : \tau \) (i.e., the structure of bound atoms and the free references are unchanged),
- \( \sigma(\nu) =_B \nu' : \tau \) (i.e., the exported binders differ by \( \sigma \)),
- \( \text{dom}(\sigma) = \text{fb}(\nu, \tau) \), \(^2\) and
- if \( \text{exported-from-} \nu \text{-top?} \) is true, \( \nu =_B \nu' : \tau \) (i.e., the exported binders are identical, and thus \( \nu =_\alpha \nu' : \tau \)),
- but if \( \text{exported-from-} \nu \text{-top?} \) is false, the exported binders of \( \nu' \) are fresh.

We omit the mechanics of threading environments of names through freshen-subterm, as they are routine. Informally speaking, “where \( \nu \) 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 \( \nu_{\text{top}} \). This ensures that \( \nu_{\text{top}} =_B \nu'_{\text{top}} : \tau_{\text{top}} \).

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

\[ \text{freshen-subterm}(\cdot|\cdot|\cdot) : \text{Value} \times \text{Type} \times \text{Bool} \rightarrow \text{Value} \times \text{Subst} \]
\[ \text{freshen-subterm}(a : \text{RAtom}, \cdot|\cdot) \triangleq a, \emptyset \]
\[ \text{freshen-subterm}(a : \text{BAtom}, \text{true}) \triangleq a, \{ \langle a, a \rangle \} \]
\[ \text{freshen-subterm}(a : \text{BAtom}, \text{false}) \triangleq aa, \{ \langle a, aa \rangle \} \text{, where } aa \text{ is fresh} \]

Given a wide product \( \text{prod}(v_i)_i \), we first recursively call freshen-subterm on all of the subterms \( v_i \), producing values \( v'_i \) whose binders have been renamed according to \( \sigma_j \).

Observe that \( v_j \) exports from \( \nu_{\text{top}} \) iff \( \text{prod}(v_i)_i \) does and \( \nu_{\text{top}} \) 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 \( j \), the value \( \llbracket \beta_j \rrbracket (\sigma_j)_j \).

Finally, we have to determine what substitution corresponds to the difference in exported binders between \( \text{prod}(v_i)_i \) and the value we return. This is equal to \( \llbracket \beta_a \rrbracket (\sigma)_j \).

\(^1\)For the sake of correctness, it is only necessary for \( \nu'_{\text{top}} \) (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.

\(^2\)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 \( \llbracket \rrbracket \).
freshen-subterm \((\text{prod} \ (v_i))_i : \text{Prod}^\beta_{\text{ex}} (\tau_i)_{\beta_i = 1} \land \text{exported-from-v-top})\)

\[
\prod \left( \left[ \beta \right] \ (\sigma_j)_j \ (v'_i : \tau_i) ight)_{i}, \ \left[ \beta_{\text{ex}} \right] \ (\sigma_j)_j
\]

where \(\forall j. (v'_j, \sigma_j) \triangleq \text{freshen-subterm} (v_j : \tau_j, \text{exported-from-v-top} \land (i \notin \beta_{\text{ex}}))\)

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 \text{exported-from-v-top} set to \text{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 \(\text{fb}(v_{\text{top}}, \tau_{\text{top}})\)).

\[
\text{freshen} \ (\_ : \_ ) : \text{Value} \times \text{Type} \rightarrow \text{Value}
\]

\[
\text{freshen} \ (v_{\text{top}} : \tau_{\text{top}}) \triangleq v'_{\text{top}}
\]

where \(v'_{\text{top}, \_} \triangleq \text{freshen-subterm} (v_{\text{top}} : \tau_{\text{top}}, \text{true})\)

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

**Conjecture 5.1** (The freshening algorithm is sound).

If \(v' = \text{freshen} \ (v : \tau)\), then \(v =_\alpha v' : \tau\) and \(A \vdash \text{suff-disj} v' : \tau\).

**Conjecture 5.2** (Freshening is always possible).

For any \(v : \tau\) and finite \(A\), there exists \(v'\) such that \(v =_\alpha v' : \tau\) and \(A \vdash \text{suff-disj} v' : \tau\)

**Proof.** Let \(v' = \text{freshen} \ (v : \tau)\). Then the result follows from Conjecture 5.1 and the totality of freshen.

This shows that it is possible to construct a value \(\text{prod} \ (v_i)\) that satisfies the \(\vdash \text{suff-disj}\) and \(=_\alpha\) premises of \(E\text{-Open}^\star\), and thus it is possible to execute Romeo programs.
Chapter 6

Romeo respects $\alpha$-equivalence: a guide to the proofs

We are now ready to prove our main theorem, which states that Romeo respects $\alpha$-equivalence. Romeo is nondeterministic in its choice of names in the E-FRESH-* and E-OPEN-* rules. This complicates the proof of respecting $\alpha$-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 $\alpha$-equivalence classes.
- Rely on the freshness condition for binders [19] to show that each manipulation on machine states (defined in terms of $\alpha$-equivalence-class representatives) respects $\alpha$-equivalence.

In such a system, we would have needed to prove only Lemma D.8 (No Names Made Up, pg. 87), as does Pottier [17]. 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 $\alpha$-equivalent environments both terminate, then their results are $\alpha$-equivalent, and second, if the evaluation of an expression in one of two $\alpha$-equivalent environments yields a result, then evaluating the expression in the other one must yield at least one $\alpha$-equivalent result as well. Since we are using big-step semantics, we cannot talk directly about non-termination. For both theorems, the vast majority of the complexity is contained in the cases for E-FRESH-* and E-OPEN-*.

Complete proofs of the theorems and lemmas are found in appendix C. 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 hand-written.

6.1 $\alpha$-equivalent environments yield $\alpha$-equivalent results

The first, and hardest, part of proving Romeo’s soundness is Theorem C.1 (pg. 77), which shows that two terminating executions of $\alpha$-equivalent environments yield $\alpha$-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-* Therefore, even though the environments start out $\alpha$-equivalent, they will not stay $\alpha$-equivalent throughout execution. For example, a fresh statement nondeterministically 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 \( \alpha \)-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 \( \sigma \circ \rho =_\alpha \sigma' \circ \rho' : \Gamma \) for some pair of injective substitutions \( \sigma \) and \( \sigma' \), then the results will be \( \alpha \)-equivalent, modulo the same transformation (i.e., \( \sigma(w) =_\alpha \sigma'(w') : \tau \)). Since we must account for faulting, we extend the definition of \( \alpha \)-equivalence to assert that \( \text{FAULT} =_\alpha \text{FAULT} \).

Consider the case of \text{E-Fresh}\(\ast\). 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 C.76–C.77, pg. 78), for the sake of the induction hypothesis.

The induction hypothesis tells us that the results (of the recursive evaluation of \( \varepsilon \)) are equivalent modulo the extended substitutions (line C.82). The result of the \text{fresh} evaluation step is either the same as that of the recursive step, or \text{FAULT}. We first show that the original substitutions suffice to \( \alpha \)-equate those two values (lines C.88–C.89), and then that one side faults if and only if the other side does (lines C.84–C.85).

The \text{E-Open}\(\ast\) case proceeds with a similar structure.\(^1\) 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 C.42 (pg. 73), used on lines C.101–C.104, described in section 6.1.1) states that the \( \vdash_{\text{suff-diag}} \) predicate in Romeo’s execution rules is strong enough that the technique from the \text{E-Fresh}\(\ast\) rules works for \text{open}, even though the various subterms of the value being destructured have potentially different scopes. In a sense, this 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, \text{E-Open}\(\ast\) proceeds like \text{E-Fresh}\(\ast\).

The \text{E-If}\(\ast\) 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 C.42 (pg. 73) 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 programatically 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. It shows that destructuring is safe if \( A \vdash_{\text{suff-diag}} \text{prod} (\{v_i\}) : \tau \) holds (where \( A \) is the set of atoms in the environment). For the purposes of Theorem C.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).

The proof of this lemma starts on line C.41 (pg. 74), which generates, not a single pair of substitutions, but one pair for each subterm of the pair of values. Lines C.44–C.45 adjust those substitutions to avoid free names in the environment, and lines C.46–C.52 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 C.53–C.55 show that all those substitutions can be combined to form a single pair of substitutions. Examining one side of that pair, lines C.56–C.62 show that that single substitution is an adequate replacement for any of the substitutions generated on line C.48.

---

1In fact, \text{open} can be used to replace \text{fresh} (whenever at least one binding form exists in the environment). We keep both constructs because we found it easiest to understand the \text{E-Open}\(\ast\) cases of our proofs a generalization of the \text{E-Fresh}\(\ast\) cases.
Lines C.64–C.65 show that, for exported subterms, their exclusion from the substitutions means that they remain $\equiv_B$ even after being substituted.

Lines C.66–C.67 show that, for subterms that are not exported, the overall pair of substitutions makes them $\equiv_B$.

### 6.1.2 Alpha-equivalence and free names are connected

Lemma C.30 (pg. 69) (along with its cases Lemmas C.28 and C.29, and its corollary, Lemma C.31) is used extensively in our proof, as it links the concepts of free names and $\alpha$-equivalence. It states that a name is free if and only if a (naive) substitution for that name can lead to a non-$\alpha$-equivalent value.

The case for binders (Lemma C.29) is straightforward, but the case for references (Lemma C.28) is much more complex. Its sublemma, lines C.7–C.12, 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 C.13–C.14), (b) $a$ being non-free, which turns into a case analysis for each subterm: (b.i) $a$ not being imported (lines C.17–C.18), and (b.ii) $a$ being imported. This final subcase does not use the sub-lemma, but instead shows that the substitutions we generated in line C.16 prove reference-equivalence of the whole value.

### 6.2 Termination is representation-oblivious

Theorem C.1 (pg. 77) leaves open the possibility that for a $\rho$ whose execution leads to a result $w$, execution of some $\alpha$-variant $\rho'$ on the same program might fail to terminate. However, Theorem C.2 (pg. 83) says that if one $\alpha$-variant terminates (either with a value or FAULT), then the execution of every $\alpha$-variant will terminate (and, by Theorem C.1, when it does, the value will be $\alpha$-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 C.2 is that, for every choice of fresh name in the original computation, (i.e., while executing fresh or open (presumably using the algorithm in chapter 5)) we choose the same name in the other one. This preserves $\alpha$-equivalence of $\rho$ and $\rho'$ for the induction hypothesis.
Chapter 7

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 D.1).

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

We use \( H, P, \) and \( C \) to range over constraints. Typically, \( H \), the hypothesis, contains facts (about the atoms in the environment) that are true by construction. \( P \), the postcondition, contains predicates that describe the connection between atoms in the environment and atoms in the output. \( C \) is used for 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 solver. (For example, Romeo-L [14] uses Z3 [4]. We discuss this in more detail in section 9.7.)

We begin by adding constraint annotations to Romeo, and giving the syntax of constraints.

\[
\begin{align*}
    fD \in \text{Fndef} & := (\text{define-fn} (f x : \tau \ldots \text{pre} C) : \tau e \text{ post} C) \\
    e \in \text{Expr} & := (f x \ldots) \\
                   & \mid (\text{fresh} x \text{ in} e) \\
                   & \mid (\text{let} x \text{ where} C \text{ be} e \text{ in} e) \\
                   & \mid (\text{case} x (x e) (x e)) \\
                   & \mid (\text{open} x (x \ldots) e) \\
                   & \mid (\text{if} x \text{ equals} x e e) \\

    z \in \text{ConstrSetVar} & := x \mid \cdot \\
    s \in \text{SetDesc} & := \emptyset \\
                   & \mid s \cup s \\
                   & \mid s \cap s \\
                   & \mid sf(z) \\
                   & \mid \mathcal{F}_r(\Gamma) \\
                   & \mid \mathcal{X}_r(x_i)_i \\
    sf \in \text{SetFn} & := \mathcal{F} \mid \mathcal{F}_r \mid \mathcal{F}_b
\end{align*}
\]
\[
H, P, C \in \text{Constraint ::= } C \land C \\
| s = s \\
| s \not= s \\
| s \# s \\
| s \subseteq s \\
| z \equiv z_{\text{qlit}} \\
| \text{true}
\]

\[
\Gamma_{\text{dot}} \in \text{TypeEnvWithDot ::= } \epsilon \mid \Gamma_{\text{dot}}, z : \tau
\]

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 \(F\), free references \(F_r\), free binders \(F_b\), and exposable names \(X_{\tau}\) of values (here, the corresponding types can be retrieved from the environment), and the free names \(F_e\) of environments (which is syntax sugar for a union of \(F_s\)), and then by the standard set constructors (\(\cup\) and \(\cap\)). Atomic formulas denote equality, inequality, etc., of sets, and \(\sim\) denotes that two values have the same free binders and references. Finally, constraints are conjunctions of atomic formulas.

The syntax for the \(X\) set function is unusual; 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 \(F, F_r,\) and \(F_b,\) the result of \(X\) 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 \(\sim\) have identical behavior under all the set functions. Furthermore, this syntax fits the way that \(X\) is used in \(\text{P·O·P}\).

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

In general, our rules are patterned after those in Pottier [18], using the type information in \(\Gamma\) to collect information about values.\(^1\)

Most rules discharge their proof obligations by delegating them to proof obligations on subexpressions. The base cases of this recursion are \(\text{P·CALL}\) and \(\text{P·QLIT}\), which describe proof obligations of the form \(\Gamma_{\text{dot}} \vdash H \Rightarrow P\).

\(\text{P·QLIT}\) 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 \(H\). \(\text{P·CALL}\) has two obligations; first, that the invoked function’s precondition is true (given \(H\)), and second that the resulting value satisfies the constraints in \(P\) (given \(H\) and the postcondition of the function).

As one might expect, the key rules are \(\text{P·FRESH}\) and \(\text{P·OPEN}\), whose definitions are closely connected to \(\text{E·FRESH}·\text{Ok}\) and \(\text{E·OPEN}·\text{Ok}\).

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 \(\text{FAULT}\) instead, and property (b) was established by the constraints imposed on the names exposed in \(\text{E·FRESH}\odot\) and \(\text{E·OPEN}·\ast\).

Now, for the purposes of the deduction system, property (a) appears in the postcondition of both \(\text{P·FRESH}\) and \(\text{P·OPEN}\), 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 \(\text{FAULTs}\). On the other hand, property (b) is a guarantee

\(^1\)Our \(\Gamma\) is similar to Pottier’s \(\Delta\).
\[
\begin{align*}
\text{typeof} (\Gamma, e_{\text{qlit}}) &= \tau & \Gamma, \cdot \vdash H \land (\cdot \equiv e_{\text{qlit}}) \Rightarrow P & \text{P-QLit} \\
\Gamma \vdash\text{proof} \{H\} \ e_{\text{qlit}} \ \{P\} \\
\text{rettype}(f) &= \tau & \text{formals}(f) = (x_{\text{formal},i}), \\
\text{argtype}(f) &= (\tau_{\text{formal},i}), & \Gamma \vdash \text{pre}(f) [x_{\text{actual},i}/x_{\text{formal},i}] & \Rightarrow P \\
\Gamma, \cdot \vdash H \land F(\cdot) \subseteq F_e((x_{\text{actual},i}:x_{\text{formal},i})) \land \text{post}(f) [x_{\text{actual},i}/x_{\text{formal},i}] & \Rightarrow P & \text{P-CALL} \\
\end{align*}
\]

\[
\begin{align*}
x \text{ fresh for } \Gamma, H, P & \quad \Gamma, x:\text{BAtom} \vdash\text{proof} \{H \land F(x) \# F_e(\Gamma)\} e \ \{P \land F(x) \# F(\cdot)\} & \text{P-FRESH} \\
\forall i. \ x_i \text{ is fresh for } \Gamma, H, P & \quad \Gamma(\text{xobj}) = \tau = \text{Prod}^{g \beta_k} (\tau_i \downarrow \beta_i), \\
\Gamma, (x_i:\tau_i) \vdash\text{proof} \{H \land \mathcal{X}_r(x_i) \# F_e(\Gamma) \land \text{xobj} \equiv (\text{prod}^{g \beta_k} (x_i \downarrow \beta_i))\} e \ \{P \land \mathcal{X}_r(x_i) \# F(\cdot)\} & \text{P-OPEN} \\
x \text{ fresh for } \Gamma, H, P, C & \quad \Gamma, x:\text{val} \vdash\text{proof} \{H \land C [x \vdash] \land F(x) \subseteq F_e(\Gamma)\} e_{\text{body}} \ \{P\} & \text{P-LET} \\
\Gamma(\chi) = \chi_0 + \tau_1 & \quad \Gamma, x_0:\chi_0 \vdash\text{proof} \{H \land \chi \equiv (\text{inj}_0 x_0 \chi_1)\} e_0 \ \{P\} \\
\Gamma, x_1:\chi_1 \vdash\text{proof} \{H \land x \equiv (\text{inj}_1 \chi_0 x_1)\} e_1 \ \{P\} & \text{P-CASE} \\
\Gamma \vdash\text{proof} \{H \land F(x_0) = F(x_1)\} e_0 \ \{P\} \\
\Gamma \vdash\text{proof} \{H \land F(x_0) \# F(x_1)\} e_1 \ \{P\} & \text{P-IfEq} \\
\Gamma \vdash\text{proof} \{H\} \ \{P\} \\
\Gamma, (x_i:\tau_i) \vdash\text{proof} \{C_0\} e \ \{C_1\} & \text{P-FnDef} \\
\vdash\text{proof} \{\text{define-fn} (f (x_i:\tau_i), \text{pre} C_0) : \tau_0 \text{ e post} C_1\} \ \{\text{ok}\} \\
\forall i. \vdash f_{\text{d}_i} \ \{\text{true}\} e \ \{\text{true}\} & \text{P-PROG} \\
\end{align*}
\]

Figure 7.1: Verification rules for the deduction system
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 \( \equiv \). This information conveys the relationship between the atoms in the scrutinee \((x_{obj} \text{ and } x_0 \text{ respectively})\) and the atoms in its component(s).

### 7.1 Odds and ends

In the \texttt{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 postcondition \( P \)) are expressed relative to the formal parameters, which are meaningless out of context. Because the actual arguments to a function invocation are all required to be variable references (rather than allowing them to be whole subexpressions), the solution is again simple: a simultaneous substitution from the formals to the actuals suffices to make the pre- and post-conditions meaningful in the caller's context.

Shadowing 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 \( \Gamma, H, \text{ and } P \); this requirement is easily satisfied by a simple renaming pass prior to type and proof checking.

P-CALL's body hypothesis contains \( \mathcal{F}(\cdot) \subseteq \mathcal{F}(\{x_{\text{actual}}, i : \tau_{\text{formal}}\}) \), a term representing extra information as a consequence of Lemma D.8, 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.

Finally, in P-IR=EQ, the information resulting from the comparison can be expressed in our predicate language (albeit in a clumsy way); 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 atom sets are disjoint.

### 7.2 Soundness of the deduction system

The soundness of the deduction system is expressed in Theorem D.1 (pg. 90), which states that a well-typed program that is accepted by the deduction system will not produce \texttt{FAULT}.

The most important lemma for proving Theorem D.1 is Lemma D.8 (pg. 87), “No Names Made Up”, which is similar to Pottier's No Atoms Made Up lemma [17]. It states that evaluation of an expression in an environment \( \Gamma \) produces a result whose free names are a subset of \( \Gamma \)'s. The proofs of Lemma D.8 and Theorem D.1 are in appendix D.

### 7.3 Example

The Romeo-L code in Figure 4.2 contains a number of \texttt{opens}, each of which potentially can produce a \texttt{FAULT}. However, our deduction shows that \texttt{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 bv. Fortunately, \( \text{inj}_{\text{lambda}} \left( \prod \text{bv.} \text{convert}(e\text{-body}) \downarrow 0 \right) \), 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.

Second, on line 7, we open up the whole \( \text{lax} \) 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 \( \text{lax} \) 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 D.8, 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 opens. 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 [18], and the same examples apply.
Chapter 8

Specified binding in Redex

Redex [7] is a toolkit “for the working semanticist”. It consists of tools for defining languages and reduction relations, to be either rendered for human consumption, or to be programmatically executed. Values in Redex are intended to resemble (the S-expression form of) human-usable object language syntax as much as possible.

Because Redex language specifications are intended to be executable, they must be formal and precise. In practice this means that, where an informal specification might say “substitution is assumed to be capture-avoiding”, the Redex semanticist must define a capture-avoiding substitution manually. Furthermore, the semanticist must remain vigilant about name collisions anywhere in code that names are manipulated.

Based on the principles in chapters 4 and 5, we added binding specifications to Redex. We observed significant decreases in code size when we adapted existing examples to our system (see section 8.3).

8.1 Differences from Romeo

Romeo uses its static types to determine the binding structure of its values. Redex values are not only untyped, but lack even runtime type information, which means that a Redex value’s (object) language must be provided as an argument to forms that manipulate it, and the role a value plays in its language is determined solely by pattern matching.

Therefore, we incorporate binding specifications as part of language definitions. The binding specifications of a language L are written as patterns in that language, augmented with a notation to indicate what names are in scope where. Even though these binding patterns are typically based on the definitions of nonterminals in the language, we write them separately because any term matching a binding pattern in L will be considered binding, regardless of what nonterminal the user intended the term to be. This is because, in Redex, term construction is nonterminal- and even language-agnostic; the user can freely take a term they have used in one language and without conversion use it in another language that has sufficiently-similar syntax.

Our implementation reflects Redex’s dynamic nature. Before a value in L is destructured, Redex will now determine what binding form it is by matching it against the various patterns defined as part of L (the pattern that will be used in destructuring is irrelevant). For example, if the extremely broad Redex pattern (any . . . ) is used to destruct the value (lambda (x) x), that value will still be determined to be a binding form. Knowing what binding form a value is, we can then freshen it by applying the algorithm in chapter 5.

For example, the lambda calculus augmented with let* (corresponding to the type Expr in figure 4.1) can be expressed as follows:
(define-language lc-with-let*
  ;; expression grammar of the language:
  (expr (lambda (x) expr)
    (expr expr)
    x
    (let* l*-clauses expr))
  ;; variables cannot be reserved words:
  (x variable-not-otherwise-mentioned)
  ;; clauses for let*:
  (l*-clauses
   (l*c x expr l*-clauses)
   no-more-clauses)
  #:binding-forms
  (lambda (x) expr #:refers-to x)
  (let* l*-clauses expr #:refers-to l*-clauses)
  (l*c x expr l*-clauses #:refers-to x)
  #:exports (shadow l*-clauses x))

Note that #:refers-to corresponds to ↓, #:exports corresponds to ↑, and shadow corresponds to ▷. It is not necessary for the binding form patterns to correspond to the grammar of the language (though they almost always will).

Also note that the syntax for let* in this language has an inconvenient feature: each clause is introduced by a distinctive identifier (l*c), and each successive clause is nested inside the previous one. This means that object language programs have to be written in an awkward syntax:

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

We address this problem in section 8.1.3.

As a further example, the event handler example from section 2.3 would be notated as follows:

(define-language language-with-handler
  (expr #| some forms omitted |#
    (handler x x expr x expr))
  (x variable-not-otherwise-mentioned)
  #:binding-forms
  #| some forms omitted |#
  (handler x_gui-elt
    x_m-evt expr_m #:refers-to (shadow x_gui-elt x_m-evt)
    x_k-evt expr_k #:refers-to (shadow x_gui-elt x_k-evt)))

Note that we are using shadowing rather than ⊎ to combine binders; because of the limited utility of ⊎,1 we have not implemented it in Redex.

### 8.1.1 A change in semantics

In the course of implementation and testing, we discovered that a small part of Romeo semantics was unacceptable for Redex. In Romeo, when a binder is neither imported nor exported by the Prod that contains it, it is considered to be a bound name (and therefore is renamed during freshening). Freshening a “bound” name that is not connected to any

---

1 Its only purpose is to identify terms as erroneous, which is better done by the metaprogrammer, who can report an appropriate error to the user.
other names is not useful, but neither was it a serious problem for Romeo. As a result of this policy, the free binders of a value would be equal to the exported binders of that value. Furthermore, for any value with free names \( v \), there would be a value without free names \( v' \) such that \( v \) would be a subterm of \( v' \). (Neither of these invariants is formally useful to Romeo; they just motivated the arbitrary decision that unused binders should be bound.)

However, in Redex, metafunctions (analogous to Romeo’s functions) internally represent their arguments as a single list value, and their parameters as a single list pattern. This means that the arguments to a function would be gathered into a larger value (without any binding specification), which would be freshened in the process of destructuring it into the metafunction’s parameters.

Suppose for example that (in the language lc-with-let*) we have \( v=(\text{l*c } x 0 \text{ no-more-clauses}) \) and \( v'=(+ x 1) \), perhaps from destructuring a \text{let*}. If we were to pass both as arguments to a metafunction, they would be treated as a bound name, and thus in need of freshening when the value is destructured. Thus when the list passed to the metafunction is unpacked, the result is two values, \( v_{\text{fresh}}=(\text{l*c } x 13846 0 \text{ no-more-clauses}) \) (perhaps), and \( v'_{\text{fresh}}=(+ x 1) \) (the reference \( x \) was free and therefore was ineligible to be freshened).

Therefore, in Redex, we treat binders that are neither imported nor exported out of \( v \) as free in \( v \). In the above example, that means that both \( x \)s are free variables, and neither is affected by the destructuring of the list, so Redex metafunction calls behave as if each argument were passed separately. In addition to fixing the immediate problem of passing arguments, this change makes values that don’t import or export any of their subterms less like binding forms: freshening such values is always a no-op.

### 8.1.2 A weakened safety guarantee

Without a type system or other static annotations, adapting Romeo’s deduction system (described in section 7) to Redex is impossible. Furthermore, implementing the escape checking that triggers errors in the OPEN-ESCAPE rule would impose a significant performance cost. Thus, the guarantees provided by our system in Redex are weaker than in Romeo. However, we will argue that this weaker soundness will still capture the essence of hygene.

Our weakened safety guarantee is that \( \alpha \)-equivalent inputs lead to outputs that are \( \alpha \)-equivalent after performing an injective renaming of its free names (we will call this “renamed \( \alpha \)-equivalent”). This weaker guarantee eliminates the need for a static deduction system, which would run contrary to the spirit of Redex. Because Redex is a complex environment, it would be instructive to consider what would happen if we made the analogous change to Romeo. In such a case, we conjecture that Theorem C.2 would still hold (i.e., termination behavior would still be unaffected by \( \alpha \)-conversion), and that Theorem C.1 would have to be weakened to the following:

**Conjecture 8.1** (Weakened determinism up to \( \alpha \)-equivalence, termination-insensitive version).

\[
\begin{align*}
\text{If} & \quad \tau = \text{typeof } (\Gamma, e) \\
\text{and} & \quad \rho =_\alpha \rho': \Gamma \\
\text{and} & \quad \Gamma \vdash_{\text{exe}} (e, \rho) \rightarrow^k w \\
\text{and} & \quad \Gamma \vdash_{\text{exe}} (e, \rho') \rightarrow^{k'} w' \\
\text{then} & \quad w =_\alpha \sigma |_{\Delta_{\rho}}(w' : \tau) : \tau,
\end{align*}
\]

where \( \sigma |_{\Delta_{\rho}}(w' : \tau) \) performs the substitution \( \sigma \) on the free names in value \( w \) at type \( \tau \). Compare this to Theorem C.1, which simply states that \( w =_\alpha w' : \tau \).
There are two things to note about this guarantee:

- In normal use cases, most names in the output are not free. Non-free names are treated the same way in the weakened and original guarantees.
- Those names in the output that are free will still (in the weaker safety guarantee) retain the same equality/inequality relative to each other. For example a program that emits the triple \((a\ b\ b)\) may also have nondeterministically emitted \((b\ c)\), but may not have emitted \((a\ a\ b)\).

To observe that our weakened safety guarantee is “nearly as strong” as the original one, imagine writing a program that takes a value (or collection of values) with free names, and wraps it in (say) a stack of lambdas, binding each free name in order of appearance (in the above example, the two results mentioned would be converted to \((\text{lambda } (a)\ (\text{lambda } (b) (a\ b\ b)))\) and \((\text{lambda } (b) (\text{lambda } (c) (b\ c\ c)))\)). This program doesn’t destroy any information about the structure of its inputs, and yet, because it eliminates free names, it produces output whose behavior is identical under both the original guarantee and the weakened one.

It is worth noting that this change relative to Romeo is in some sense “unwinding” the change Pure FreshML [17] makes relative to FreshML [21]. Our conjectured safety guarantee is slightly stronger than FreshML’s Correctness of Representation Theorem (their Theorem 5.6), which only states that termination behavior is equivalent, but we believe that an equivalent of Conjecture 8.1 also holds for their system.

Other correctness issues

Redex built in the Racket programming language, and Redex programmers have the option of using arbitrary Racket code while defining Redex models. Because the safety guarantee provided by our system requires freshening terms whenever they are destructured, binding safety is only provided if the user refrains from writing Racket code that examines terms. In the future, Redex may switch to a nearly-opaque representation of terms, making it nearly impossible to accidentally write code that violates the safety guarantee.

On the other hand, under some circumstances, it is desirable to violate binding safety (even weak binding safety): for example, a compiler should produce error messages in terms of the variables the programmer actually wrote, even if those variables were bound.

Finally, we had to make a change to behavior of Redex for correctness’s sake. Redex metafunctions are assumed by default to be deterministic, and so metafunction invocations are memoized. In rare cases, the nondeterminism caused by freshening is exposed as nondeterminism in the output of a metafunction (this coincidentally corresponds to code that would have produced a fault, had the safety guarantee not been weakened). To avoid incorrect behavior, we have disabled memoizing the result of a metafunction whenever its execution involves freshening.

8.1.3 Binding specification for ellipses

Unlike the Romeo pattern-matcher, Redex pattern-matching is quite powerful. Patterns may be nested arbitrarily deeply, and may contain the ellipsis notation “...” for repeated sub-patterns. Furthermore, it is possible for the same name to appear multiple places, indicating that the corresponding subterms must be identical.

All of these features have relatively straightforward implementations; our system integrates well with established pattern-matching techniques. But one feature deserves a more detailed explanation: binding along ellipses.

When a subterm of a binding form is syntactically repeated, there is no natural way to express binding relationships between those repetitions. Such a result is semantically achievable by restructuring the repetition into a separate recursive binding form, but,
in the system of section 8.1, every binding form must be distinguishable from all other language forms. In practice, this means reserving a keyword to be used in every repeat of the subform. Since our goal is to enable safe manipulations of surface syntax, this is unacceptable.

Instead, we introduce a variation of . . . , called #: . . . bind, which matches the same syntax, but imposes binding structure onto it as if it were a separate binding form. For example, a version of let* with natural syntax can be defined with the following notation:

```scheme
(let* ((x expr) #: . . . bind (clauses x (shadow clauses x)))
  expr_body #:refers-to clauses)
```

Syntactically, (x expr) can be repeated arbitrarily many times. A #: . . . bind takes three arguments. The first argument (the “name”; in this case, clauses) allows the whole form to refer to repetition as a whole (e.g., the #:refers-to clauses in this example), and for individual repetitions to refer to all the subsequent repetitions. The second argument is a β (the “import”; in this case, x), which is imported into all subsequent repetitions. The third argument is also a β (the “export”; in this case, (shadow clauses x)), which controls what happens when the name argument is mentioned a β, just like the export of an independent binding form.

Our running example is now a valid term without any processing:

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

In the middle clause (the one that introduces the name b), x refers to b and expr refers to (+ a a). According to the export argument of the #: . . . bind, the names c (from the remaining clause, named clauses by the name argument) and a are exported. Furthermore, according to the import argument, the name b will be bound in the remaining clause.

### 8.2 Translation of let*

It is now possible to express our example of Romeo code that desugars a natural syntax for let* (from Figures 4.1 and 4.2) as a Redex metafunction, which we do in Figure 8.1.

### 8.3 Improving existing Redex models

Redex comes with a number of example models. We examined several of the models that had interesting binding structure.

#### 8.3.1 Church numerals

church.rkt contains a demo of Church numerals, in a basic lambda calculus augmented with let. The original version of the file (in Figure 8.2) contains 44 lines of code, but when binding specifications are added (in Figure 8.3), the large and complex subst metafunction is removed, reducing the number of lines of code to 29. This dramatic change is a result of the presence of two binding forms (let and lambda) in an otherwise extremely simple language.

---

2When we say “lines of code”, we refer to lines that contain at least one nonblank character, excluding tests and comments.
(define-language lc-with-let*
  (x variable-not-otherwise-mentioned)
  (e x
   (e e)
   (lambda (x) e)
   (let* ([x e] ...) e))
  #:binding-forms
  (lambda (x) e #:refers-to x)
  (let* ([x e] #:...bind (clauses x (shadow clauses x)))
   e_body #:refers-to clauses))

(define-metafunction lc-with-let*
  desugar : e -> e
  [(desugar x) x]
  [(desugar (e_l e_r)) ((desugar e_l) (desugar e_r))]
  [(desugar (lambda (x) e)) (lambda (x) (desugar e))]
  [(desugar (let* () e)) (desugar e)]
  [(desugar (let* ([x e_val] [x_rest e_val-rest] ...) e_body))
   ((lambda (x)
      (desugar (let* ([x_rest e_val-rest] ...) e_body)))
    (desugar e_val))])

Figure 8.1: Desugaring of let* in Redex

8.3.2 Recursive let

letrec.rkt demonstrates the semantics of letrec. It uses the subst.rkt library to
perform capture-avoiding substitution. However, subst.rkt assumes that the language
it is operating on only has λ as its only binding form, and so a comment at the top of
letrec.rkt reads:

BUG: letrec & let are not handled properly by substitution

This bug is easily fixed by adding binding specifications, as shown in Figure 8.4. Since the
(partially) capture-avoiding substitution was provided by an external library, the total
number of lines barely changes: it goes from 93 to 91.

Two things are worth noting about this model. The first is that even forms that aren’t
expressions, like ((store (x v) ...) e), can have binding structure. A consequence
of this is that we can recognize the α-equivalence of the program states ((store (a 1))
a) and ((store (b 1)) b).

The second noteworthy thing is that this model contains an example of how it does
not suffice for binding-safe programming to merely construct a capture-avoiding substi-
tution. In the original version’s reduction rule for let, the programmer had to freshen a
name before adding it to the store (as in Figure 8.5), but with binding specifications, it is
safe to write the reduction rule in the natural style (Figure 8.6).

8.3.3 π-calculus

pi-calculus.rkt has a different set of binding forms than a λ-calculus-based model. In
the π-calculus, the binding forms are nu and in (see Figure 8.7).

As in the above examples, our introduction of binding forms allowed us to remove a
substitution metafunction. Furthermore, two name-related side conditions became re-
#lang racket

(require redex)
(reduction-steps-cutoff 100)

(define-language lang
  (e (lambda (x) e)
     (let (x e) e)
     (app e e)
     (+ e e)
     number)
  (e-ctxt (lambda (x) e-ctxt) a-ctxt)
  (a-ctxt (let (x a-ctxt) e) (app a-ctxt e) (app x a-ctxt) hole)
  (v (lambda (x) e) x)
  (x variable))

(define reductions
  (reduction-relation lang
    (--> (in-hole e-ctxt_1 (app (lambda (x_1) e_body) e_arg))
      (in-hole e-ctxt_1 (subst (x_1 e_arg e_body))))
    (--> (in-hole e-ctxt_1 (let (x_1 v_1) e_1))
      (in-hole e-ctxt_1 (subst (x_1 v_1 e_1))))))

(define-metafunction lang
  [[(subst (x_1 e_1 (lambda (x_1) e_2))) (lambda (x_1) e_2)]]
  [(subst (x_1 e_1 (lambda (x_2) e_2)))
    ,(term-let ((x_new (variable-not-in (term e_1) (term x_2))))
      (term (lambda (x_new) (subst (x_1 e_1 (subst (x_2 x_new e_2)))))))]
  [[(subst (x_1 e_1 (let (x_1 e_2) e_3))) (let (x_1 (subst (x_1 e_1 e_2))) e_3)]
    [(subst (x_1 e_1 (let (x_2 e_2) e_3)))
      ,(term-let ((x_new (variable-not-in (term e_1) (term x_2))))
        (term (let (x_2 (subst (x_1 e_1 e_2))) (subst (x_1 e_1 (subst (x_2 x_new e_3)))))))]
  [[(subst (x_1 e_1 x_1)) e_1]
    [(subst (x_1 e_1 x_2)) x_2]
  [[(subst (x_1 e_1 (app e_2 e_3)))
    (app (subst (x_1 e_1 e_2))
      (subst (x_1 e_1 e_3)))]
    [(subst (x_1 e_1 (+ e_2 e_3)))
      (+ (subst (x_1 e_1 e_2))
        (subst (x_1 e_1 e_3)))]
  [[(subst (x_1 e_1 number_1)) number_1]]

Figure 8.2: The original contents of church.rkt.
#lang racket
(require redex)
(reduction-steps-cutoff 100)

(define-language lang
  (e (lambda (x) e)
    (let (x e) e)
    (app e e)
    (+ e e)
    number x)
  (e-ctxt (lambda (x) e-ctxt)
    a-ctxt)
  (a-ctxt (let (x a-ctxt) e)
    (app a-ctxt e)
    (app x a-ctxt)
    hole)
  (v (lambda (x) e)
    x)
  (x variable)
#:binding-forms
  (lambda (x) e #:refers-to x)
  (let (x e) e_body #:refers-to x))

(define reductions
  (reduction-relation
   lang
   (--> (in-hole e-ctxt_1 (app (lambda (x_1) e_body) e_arg))
        (in-hole e-ctxt_1 (substitute e_body x_1 e_arg)))
   (--> (in-hole e-ctxt_1 (let (x_1 v_1) e_1))
        (in-hole e-ctxt_1 (substitute e_1 x_1 v_1))))))

Figure 8.3: church.rkt, rewritten to use binding specifications.
(define-language lang
 (p ((store (x v) ...) e))
 (e (set! x e)
    (let ((x e)) e)
    (letrec ((x e)) e)
    (begin e e ...)
    (e e)
    x
    v)
 (v (lambda (x) e)
    number)
 (x variable)
 (pc ((store (x v) ...) ec))
 (ec (ec e)
    (v ec)
    (set! variable ec)
    (let ((x ec)) e)
    (begin ec e e ...)
    hole)
 #:binding-forms
 (lambda (x) e #:refers-to x)
 (let ((x e)) e_body #:refers-to x)
 (letrec ((x e #:refers-to x)) e_body #:refers-to x)
 ((store (x v) #:refers-to (shadow x ...) ...) e #:refers-to (shadow x ...))
)

Figure 8.4: The language definitions and binding specifications for letrec.rkt

(==> ((store (name the-store any) ...) (in-hole ec_1 (let ((x_1 v_1)) e_1)))
 (let ((new-x (variable-not-in (term (the-store ...)) (term x_1))))
  (term
   ((store the-store ... (,new-x v_1))
    (in-hole ec_1 (subst (x_1 ,new-x e_1))))))
 let)

Figure 8.5: Without binding specifications, this reduction rule has to manually freshen a variable.

(==> ((store (name the-store any) ...) (in-hole ec_1 (let ((x_1 v_1)) e_1)))
 ((store the-store ... (x_1 v_1)) (in-hole ec_1 e_1)
 let)

Figure 8.6: With binding specifications, the natural version of the code is safe.

46
Figure 8.7: The π-calculus, with binding annotations.

dundant and were removed. The overall number of lines of code changed from 144 to 90 with our changes.

8.4 Benchmarks

8.4.1 Microbenchmarks

We conducted microbenchmarks to assess the performance of various components of our system, and to compare them to both “naive” (non-binding-respecting) and “manual” (hand-rolled, specific to the language) versions of the same tasks. In some cases, we were also able to measure the performance after “tagging” the relevant values with their binding specification, thereby isolating the cost of performing the task from the task of determining what binding form the relevant value is.

The test language is so simple that its binding forms are nonrecursive. The version with binding specifications is defined as follows:

```
(define-language binding:L
  (e (bd x x)
      (complex-bd ((x x) ...) ... x))
  (x variable-not-otherwise-mentioned)
  #:binding-forms
  (bd x x_body #:refers-to x)
  (complex-bd ((x_0 x_v #:refers-to x_0) ...) ...
              x_body #:refers-to (shadow (shadow x_0 ...) ...))
)
```

We use (complex-bd ((a a) (b f1) (c c)) ((d f2)) c) as the example v_c. “Comparison” is α-equivalence, except for the “naive” example, which tests for literal equivalence.
The nonrecursive nature of the language makes the implementation of manual freshening and α-equivalence extremely simple, so it is not surprising that the cost of manual operations is low.

### 8.4.2 Launchbury’s semantics for lazy evaluation

We also examined the execution time of a basic lazy language model. We compare a pre-existing system that manually achieves name safety, to a version we modified to take advantage of binding specifications.

The first task we examine is capture-avoiding substitution. In this case, substitution is implemented as a metafunction in the pre-binding-specifications implementation of Launchbury’s semantics. The overhead of Redex’s recursive metafunction calls gives our system (which has a built-in substitution function, implemented in plain Racket) an advantage, so we also tested a version in which we use a substitution metafunction whose capture-avoiding nature arises from the guarantees inherent in our system.

We test our substitution on values with the following forms (in each case, we are substituting the name \( x \) with a different name):

\[
\begin{align*}
v_1 & : (\lambda (y) ((\lambda (x) y) x)) \\
v_2 & : (\lambda (y) ((\lambda (q) x) x)) \\
v_3 & : (\text{let } ([x 1] [y 2] [z 3]) (+ x y)) \\
v_4 & : (\text{let } ([x (\lambda (x) (x x))] [y ((\lambda (x) (x x)) x)] [z ((\lambda (x) (x x)) x)] (+ (+ (+ (\lambda (x) (x x)) (\lambda (x) (x x))) (\lambda (x) (x x)))) (\lambda (x) (x x))) (\lambda (x) (x x))) (\lambda (x) (x x)))
\end{align*}
\]

<table>
<thead>
<tr>
<th>Task (25000 times)</th>
<th>Our system</th>
<th>Tagged</th>
<th>Manual</th>
<th>Naive</th>
</tr>
</thead>
<tbody>
<tr>
<td>Destructure (bd a a)</td>
<td>11.42 (0.28)</td>
<td>10.09 (0.21)</td>
<td>6.98 (0.10)</td>
<td>6.21 (0.11)</td>
</tr>
<tr>
<td>Destructure ( v_c )</td>
<td>35.51 (0.58)</td>
<td>19.77 (0.47)</td>
<td>9.15 (0.22)</td>
<td>7.26 (0.14)</td>
</tr>
<tr>
<td>Compare (bd a a)</td>
<td>7.081 (0.131)</td>
<td>0.182 (0.008)</td>
<td>0.039 (0.001)</td>
<td></td>
</tr>
<tr>
<td>Compare ( v_c )</td>
<td>49.090 (0.335)</td>
<td>0.751 (0.016)</td>
<td>0.074 (0.004)</td>
<td></td>
</tr>
</tbody>
</table>

Mean (and standard deviation) of ten trials. All times are in seconds.

The value \( v_4 \) was chosen to have many identical subterms and thereby take advantage of metafunction caching, to measure the cost of the fact that we must disable metafunction caching in some cases. (See section 8.1.2.)

Finally, we examine the performance of some specific programs in this language.
\( \epsilon_{\text{add-5}}: \) (let ([Y (λ (f)
  (let ([g (λ (x) (let ([xx (x x)])
    (f xx)))]
    (g g)]))
  [tri (λ (me)
    (λ (x)
      (if0 x
        0
        (let ([x1 (+ x -1)])
          (+ (me x1) x)))))]
  [five 5])
  ((Y tri) five))

\( \epsilon_{\text{noop}}: \) (let ([tri (λ (x)
  (let ([x1 (+ x -1)]) x)])
  [five 5])
  (tri five))

\( \epsilon_{\text{awkward-add}}: \) (let ([o 1] [t 2] [r 3] [f 4])
  (((((λ (x) (λ (y) (λ (z) (λ (w)
    (+ (+ x y) (+ w z)))))
    o) t) r) f))

<table>
<thead>
<tr>
<th>Task</th>
<th>Our system</th>
<th>Manual</th>
</tr>
</thead>
<tbody>
<tr>
<td>Run ( \epsilon_{\text{add-5}} ) 200 times</td>
<td>28.16 (0.41)</td>
<td>19.85 (0.24)</td>
</tr>
<tr>
<td>Run ( \epsilon_{\text{noop}} ) 1000 times</td>
<td>29.50 (0.73)</td>
<td>30.59 (0.40)</td>
</tr>
<tr>
<td>Run ( \epsilon_{\text{awkward-add}} ) 500 times</td>
<td>41.88 (0.79)</td>
<td>31.39 (0.52)</td>
</tr>
</tbody>
</table>

Mean (and standard deviation) of ten trials. All times are in seconds.

In two of the examples, the performance advantage of the built-in substitution function is completely overshadowed by the cost of performing freshening every time the implementation performs pattern matching. However, the overall cost of binding-safe program is small compared to the overall cost of execution.

### 8.4.3 Church numerals

We tested the performance effect of our change to the Church numeral example (section 8.3.1) in two ways. First, we evaluated a large expression that defined addition, multiplication, and the numbers 1 and 2 in a Church encoding, and calculated \((2 + 2) \times 2 \times 1\).

Second, we used Racket’s dynamic-require feature to measure the time required to compile the Redex code defining the two models.

<table>
<thead>
<tr>
<th>Task</th>
<th>Our system</th>
<th>Manual</th>
</tr>
</thead>
<tbody>
<tr>
<td>Evaluate large expression</td>
<td>96.4 (4.7)</td>
<td>127.6 (6.5)</td>
</tr>
<tr>
<td>Compile the model</td>
<td>680.49</td>
<td>969.59</td>
</tr>
</tbody>
</table>

Mean (and standard deviation) of ten trials. All times are in milliseconds.

In the evaluation tests, our implementation shows that the efficiency of a purpose-built substitution function (in this case) outweighs the cost of the extra freshening required every time a value is destructured.

The compilation tests show a result that may be more practically useful: the substantially decreased code size means shorter compilation times. For a semanticist, decreasing the cost of repeated recompilation may improve the experience of model development.
8.4.4 Recursive let

We also tested the performance effect of our changes to the recursive let model. The existing model lacked tests, so we wrote a set of tests for the purposes of benchmarking. Note that this comparison is in some sense apples-to-oranges: the original version ("Manual") is based on a buggy substitution that doesn’t account for all the binding forms in the language.

<table>
<thead>
<tr>
<th>Task</th>
<th>Our system</th>
<th>Manual</th>
</tr>
</thead>
<tbody>
<tr>
<td>Run the test suite</td>
<td>1887 (51)</td>
<td>1181 (40)</td>
</tr>
<tr>
<td>Compile the model</td>
<td>902 (34)</td>
<td>907 (37)</td>
</tr>
</tbody>
</table>

Mean (and standard deviation) of one hundred trials. All times are in milliseconds.

In this example, in contrast to the church numerals example, we can see a significant cost to evaluating in Redex models with binding specifications. As expected, there is no significant difference in compilation time, since the original version of the model delegates its definition of capture-avoiding substitution to a library.
Chapter 9

Related work

9.1 Statically specified binding in template macros

The work of Herman and Wand [10, 11] 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 [18]. Both Pure FreshML and Romeo are first-order, side-effect-free languages in which a runtime system ensures that introduced names do not escape their scope, and both provide a proof system that generates proof obligations which, if true, guarantee statically that no faults will occur. One 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 Comil [16]. 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 [20] 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 C.1 and C.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 [5], 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.

The original attempts at specification, given by Clinger [3] and Dybvig [5], 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 [1] presents a plausible definition that examines the support of unexpanded code, while being agnostic as to its binding structure. A reformulation by Matthew Flatt [8] called “sets of scopes” provides largely the same behavior with a simpler semantics and in an easier-to-use fashion. However, we feel that the use of static binding specifications (introduced for this purpose by Herman [10]) 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 the static guarantee that name collisions cannot occur, rooted in the binding specifications of the terms that it manipulates.

9.5 Binding in theorem-proving systems

Ever since the POPLMark Challenge [2], there has been a large interest in coding terms with bindings in various proof assistants [15, 19, 22]. 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 [23] 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 `⟨\prod e\, p\, e\, \downarrow_0⟩`.

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 Coq [16], but extended to the point that it supports constructs like `let*`. Like Coq, it is oriented towards abstract syntax, not concrete syntax like Romeo.

9.7 Extensions to Romeo

As we have described it, writing programs in Romeo is tedious. Programs must be written in a monadic (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 $\Gamma \vdash H \Rightarrow P$ is to be determined, and, if it fails, how the programmer is supposed to figure out how to fix it.

Romeo-L also includes a connection to the Z3 SMT solver [4], which is able to check statements of the form $\Gamma \vdash H \Rightarrow P$, 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 $0$, $1$, $\leq 1$, or $\geq 1$). At that point, the constraints are directly expressible as SMT problems using combinatory array logic (CAL) 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`. 
Chapter 10

Future work

It is our hope that this work can inspire others to make safe metaprogramming more possible. Here, we speculate about forms that work might take.

10.1 Binding features

Although the binding specification features we describe give the syntax designer a great deal of freedom in the structure of binding forms, there are many aspects of binding that we have not examined. Here, we speculate about how these other aspects might be implemented.

- Forms like Racket's `struct` bind names that are concatenations of names provided by the user. For example, `(struct point (x y))` defines (among others) the names `point-x` and `point-y`. Designing (and ensuring the safety of) an extension to `β` (see section 2.3) that (say) takes the cross product of two `β`s by concatenating one name from each, is (we speculate) straightforward. However, such a feature would be useless if the metaprogrammer has no way of determining which concatenated name is which! Typically, metaprogrammers can examine the binders of a binding form, but in a binding-safe system, knowing which name is `point` and which is `x` does not help the metaprogrammer identify `point-x`.

Perhaps this means that new features of `β` that can generate names must be accompanied by new functions that can regenerate those same names. Binding-safe systems are typically thought of as having no name operations besides fresh name generation and comparison of names for equality, so this is unexplored territory.

- Some names are neither properly binders or references, such as type variables in type environments used by typecheckers. Although machine configurations are not usually thought of as syntax per se, they certainly have binding structure, and would benefit from the correct treatment of names. The simplest approach would be a notation to express that all otherwise free references (with the same name) in a subterm are bound to each other While we don’t anticipate implementing this feature to be particularly complicated, our formal definition of `α`-equivalence might need to be substantially revised to accommodate this.

- Anaphoric macros bind names that are selected by the macro writer, not the macro user. For example, in the body of Common Lisp’s `loop`, the name `it` is in scope. We believe that this is a special case of the problem of name concatenation: again, the main challenge is not the extension to `β` nor the binding safety of the new feature, but allowing the metaprogrammer to know what name was bound in that way.
Languages with macro systems pose an additional challenge: macros, like values, may be defined for only a particular scope. For example, the Scheme construct `let-syntax` introduces a new macro which can be referenced only inside its body. Therefore, the body of a `let-syntax` has unknown binding structure until enough information has been extracted from its clauses to be able to identify new macros and to know what binding behavior they have.

If macros are identified by name binding in the same namespace as “normal” variables (as they are in Scheme), then this information should be treated as a form of variable binding! Thus, where Romeo manipulates sets of names, a system supporting local macro definitions would manipulate maps from names to information about their referrent’s role in binding at the syntax level.\(^1\) Then, determining the binding behavior of a term requires looking up names from the relevant map. However, without any restrictions on binding new macros, it is easy to define a syntactic form that cannot be implemented. For example, `letrec-syntax` (a form that can refer to the syntactic forms it defines), unless defined very carefully, would have subterms whose binding behavior would be impossible to determine until after their own exports are known. It seems that some acyclicness property must be enforced.

Many languages have multiple namespaces (for example, a language might separate type names from variable names). Quotative constructs may introduce a new namespace, or disable binding entirely (like Lisp’s `quote`). We believe that ad-hoc techniques can handle specific use cases, but we don’t have a clear picture of what a generalization might look like.

In the extreme case, metaprogrammers might want to be able to define the binding behavior of their terms in terms of an arbitrary computation performed on those terms (i.e., to perform arbitrary computation inside a \(\beta\)). While this might be excessively expressive, we see no theoretical obstacle to this, and it might subsume many ad-hoc extensions to binding specification systems.

Module systems have complex name behavior; in a sense, they allow a single name to stand for a large set of names, which can be unpacked selectively. As with the problem of local macro definitions, information associated with the name (of the module) when it is bound has a role on binding when the name is referenced. So similarly, we expect that extending the information carried with bound names will make the binding behavior of modules tractable.

### 10.2 Typed macros

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! Typically, programmers in such languages are advised to use macros sparingly.

A macro system in which macro definitions and macro invocations can both be typechecked before macros expansion would be a major breakthrough in the usability of macros in typed languages. SoundX [13] is an exciting step in this direction, demonstrating extensions to language syntax whose expansions are guaranteed to be type-safe. We believe that the major remaining steps between SoundX and true typed macros are:

\(^1\)This echoes the `Env` of Dybvig [5] mapping each symbol to a transformer (macro) or “Variable” (or “Special”, for core forms that are in scope by default), making information about macros available to the expander.
• Extensions defined by arbitrary code, instead of pattern matching. This is essentially the difference between $\lambda_m$ [10] 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. As described above, 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.\(^2\) We have shown that it is instead possible to automatically rename to avoid collisions, which is far more convenient to the user.

Although these features are not small, we believe that typed macros will be a major user-friendliness improvement for the interaction between types and macros.

\(^2\)Although it is not immediately obvious, SoundX does possess 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 that expression.
Appendix A

An example derivation of $\equiv_R$

c $\equiv_R$ bb:BAtom
(\ast bb 5) $\equiv_R$ (\ast bb 5):Expr
(c (\ast bb 5)) $\equiv_R$ (bb (\ast bb 5)):Prod^0 (BAtom, Expr)
b $\equiv_R$ aa:BAtom
(+ aa aa) $\equiv_R$ (+ aa aa):Expr
(b (+ aa aa)) $\equiv_R$ (aa (+ aa aa)):Prod^0 (BAtom, Expr)
(b (+ aa aa)) $\Rightarrow$ (aa (+ aa aa)):Prod^0 (BAtom, Expr)
\[\{b, bb\} \{c (\ast bb 5)\} = (c (\ast bb 5))\]
\[\{aa, bb\} \{aa (\ast aa 5)\} = (bb (\ast bb 5))\]
\[\langle (b (+ aa aa)) \rangle =R \langle (aa (+ aa aa)) \rangle :\text{LetStarClauses}\]
a $\equiv_R$ d:B Atom
1 $\equiv_R$ 1:Expr
(a 1) $\equiv_R$ (d 1):Prod^0 (BAtom, Expr)
(a 1) $\Rightarrow$ (d 1):Prod^0 (BAtom, Expr) $\rightarrow$ \{a, aa\} $\Rightarrow$ \{d, aa\}
\[\{a, aa\} \{b (+ a a)\} = \langle (b (+ aa aa)) \rangle \]
\[\{c (\ast b 5)\} = \langle (c (\ast b 5)) \rangle \]
\[\{d, aa\} \{d (+ d d)\} = \langle (aa (+ aa aa)) \rangle \]
\[\{d, aa\} \{d (+ d d)\} = \langle (aa (\ast aa 5)) \rangle \]
\[
\langle (a 1) \langle (b (+ a a)) \rangle \rangle =R \langle (d 1) \langle (d (+ d d)) \rangle \rangle :\text{LetStarClauses}\]

(display cc) $\equiv_R$ (display cc):Expr
\[\langle (a 1) \langle (b (+ a a)) \rangle \rangle =R \langle (d 1) \langle (d (+ d d)) \rangle \rangle :\text{LetStarClauses}\]
\[\langle (c (\ast b 5)) \rangle =R \langle (d (\ast d 5)) \rangle :\text{LetStarClauses}\]
\[\rightarrow \{a, aa\}, \{b, bb\}, \{c, cc\} \Rightarrow \{d, cc\}\]
\[
\{a, aa\}, \{b, bb\}, \{c, cc\} ((\text{display } c)) = (\text{display } cc) \\
\{d, cc\} ((\text{display } d)) = (\text{display } cc) \tag{A.20} \quad \text{Substitution.} \\
\{\text{let* } ((a 1) \\
\quad (b (+ a a)) \\
\quad (c (* b 5))) \} \Rightarrow \{\text{let* } ((d 1) \\
\quad (d (+ d d)) \\
\quad (d (* d 5))) \} \Rightarrow \text{display } d) \tag{A.22} \quad \text{R}_\alpha-\text{PROD, A.17, A.19, A.20, A.21, and A.18.}
\]
Appendix B

An example derivation of $\bowtie$

Define: $v_0 \triangleq ((a 1) (b (* a a)) (c (* b 5)))$ and $v_0' \triangleq ((d 1) (d (* d d)) (d (* d 5)))$

$c \bowtie d$:

\[ BAtom \rightarrow \{\langle c, cc \rangle \} \bowtie \{\langle d, cc \rangle \} \]

(B.1)

$(* b 5) \bowtie (* d 5)$:

\[ Expr \rightarrow \emptyset \bowtie \emptyset \]

No free bindings.  (B.2)

$(c (* b 5)) \bowtie (d (* d 5))$:

\[ Prod^{\bowtie 0} (BAtom, Expr) \rightarrow \{\langle c, cc \rangle \} \bowtie \{\langle d, cc \rangle \} \]

J-Prod, B.1 and B.2.  (B.3)

$b \bowtie d$:

\[ BAtom \rightarrow \{\langle b, bb \rangle \} \bowtie \{\langle d, bb \rangle \} \]

J-Atom.  (B.4)

$(* a a) \bowtie (+ d d)$:

\[ Expr \rightarrow \emptyset \bowtie \emptyset \]

No free bindings.  (B.5)

$(b (* a a)) \bowtie (d (* d d))$:

\[ Prod^{\bowtie 0} (BAtom, Expr) \rightarrow \{\langle b, bb \rangle \} \bowtie \{\langle d, bb \rangle \} \]

J-Prod, B.5 and B.6.  (B.6)

$[[1 \bowtie 0]] (\{\langle d, cc \rangle \}, \{\langle d, bb \rangle \}) = \{\langle d, cc \rangle \}$

Def. of $[[\cdot]]$.  (B.7)

$v_{0,1} \bowtie v_{0,1}'$:

\[ LetStarClauses \rightarrow \{\langle a, aa \rangle \} \bowtie \{\langle d, aa \rangle \} \]

J-Prod, B.4, B.7, and B.8.  (B.8)

$a \bowtie d$:

\[ BAtom \rightarrow \{\langle a, aa \rangle \} \bowtie \{\langle d, aa \rangle \} \]

J-Atom.  (B.9)

$1 \bowtie 1$:

\[ Expr \rightarrow \emptyset \bowtie \emptyset \]

No free bindings.  (B.10)

$(a 1) \bowtie (d 1)$:

\[ Prod^{\bowtie 0} (BAtom, Expr) \rightarrow \{\langle a, aa \rangle \} \bowtie \{\langle d, aa \rangle \} \]

J-Prod, B.10 and B.11.  (B.11)

$[[1 \bowtie 0]] (\{\langle d, cc \rangle \}, \{\langle d, aa \rangle \}) = \{\langle d, cc \rangle \}$

By def. of $[[\cdot]]$.  (B.12)

$v_0 \bowtie v_0'$:

\[ LetStarClauses \rightarrow \{\langle a, aa \rangle , \langle b, bb \rangle, \langle c, cc \rangle \} \bowtie \{\langle d, cc \rangle \} \]

J-Prod, B.12, B.9, and B.13.  (B.13)

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

Soundness of execution for $\alpha$-equivalence

C.1 General definitions

C.1.1 Naive operations

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

Lemma C.1 (Substitutions never increase disjointness).
If $\sigma(A) \# \sigma(A')$, then $A \# A'$.

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

Lemma C.2 (Substitutions can sometimes commute).
If $\text{dom}(\sigma) \# \text{dom}(\sigma')$ and $\text{dom}(\sigma) \# \text{rng}(\sigma')$ and $\text{dom}(\sigma') \# \text{rng}(\sigma)$, then $\sigma(\sigma'(v)) = \sigma'(\sigma(v))$.

Proof.

$a_d \neq a'_d$ and $a_d \neq a'_r$ and $a'_d \neq a_r$ implies $v[a_d/a_r][a'_d/a'_r] = v[a'_d/a'_r][a_d/a_r]$

$a'_r \notin \text{dom}(\sigma)$ and $a'_r \notin \text{dom}(\sigma)$ and $a'_d \notin \text{rng}(\sigma)$

implies $\sigma(v)[a'_d/a'_r] = \sigma(v[a'_d/a'_r])$

$\text{dom}(\sigma) \# \text{dom}(\sigma')$ and $\text{dom}(\sigma) \# \text{rng}(\sigma')$ and $\text{rng}(\sigma) \# \text{dom}(\sigma')$

implies $\sigma'(\sigma(v)) = \sigma(\sigma'(v))$

By calculation.

By induction on the size of $\sigma$.

By induction on the size of $\sigma$.

C.1.2 Semantics of $\beta$

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.

Lemma C.3 ($\beta$ of $A$ is consistent with its inputs).
$[\beta](A_i)_i = \bigcup_{i \in \beta} A_i$
Proof. Follows from an inductive argument on the size of \( \beta \). Each case follows from the definitions of \( i \in \beta \) and \( \llbracket \beta \rrbracket (A_i)_i \).

**Lemma C.4** (\( \beta \) of \( \sigma \) is consistent with its inputs).
\[
\text{dom}(\llbracket \beta \rrbracket (\sigma)_i) = \bigcup_{i \in \beta} \text{dom}(\sigma_i) \quad \text{and} \quad \llbracket \beta \rrbracket (\sigma)_i \subseteq \bigcup_{i \in \beta} \sigma_i
\]

**Proof.** Follows from an inductive argument on the size of \( \beta \). Each case follows from the definitions of \( i \in \beta \) and \( \llbracket \beta \rrbracket (\sigma)_i \).

**Lemma C.5** (\( \beta \) of \( \sigma \) ignores the \( \sigma \)s not mentioned by \( \beta \)).
If \( \forall i \in \beta. \sigma_i = \sigma_i' \), then \( \llbracket \beta \rrbracket (\sigma)_i = \llbracket \beta \rrbracket (\sigma_i')_i \)

**Proof.** Follows from a straightforward inductive argument on the size of \( \beta \).

**Lemma C.6** (\( \beta \) of \( \sigma \) with disjoint domains does not shadow anything).
Suppose that \( \forall i, j \in \beta, i \neq j \Rightarrow \text{dom}(\sigma_i) \# \text{dom}(\sigma_j) \). Then \( \llbracket \beta \rrbracket (\sigma)_i = \bigcup_{i \in \beta} \sigma_i \).

**Proof.** Follows from an inductive argument on \( \beta \), which is straightforward in every case except \( \beta = \beta' \bowtie \beta'' \) (and \( \cup \)), which is similar. In that case:

\[
\llbracket \beta' \bowtie \beta'' \rrbracket (\sigma)_i = \bigcup_{i \in \beta'} \sigma_i \quad \text{By IH.} \quad \text{(C.1)}
\]

\[
\langle a, a' \rangle \in \llbracket \beta' \rrbracket (\sigma)_i \quad \text{and} \quad \langle a, a'' \rangle \in \llbracket \beta'' \rrbracket (\sigma)_i \Rightarrow a' = a''
\]

\[
\llbracket \beta' \bowtie \beta'' \rrbracket (\sigma)_i = \bigcup_{i \in \beta' \bowtie \beta''} \sigma_i \quad \text{By domain disjointness and Lemma C.4.}
\]

\[
\llbracket \beta' \bowtie \beta'' \rrbracket (\sigma)_i = \bigcup_{i \in \beta'} \sigma_i \quad \text{By C.1 and the definition of \( \llbracket \beta \rrbracket (\sigma)_i \).}
\]

**Lemma C.7** (\( \beta \) of injective, mutually-range-disjoint \( \sigma \)s has an injective result).
If \( \forall i. \text{inj}(\sigma_i) \) and \( \forall i \neq j. \text{rng}(\sigma_i) \# \text{rng}(\sigma_j) \), then \( \text{inj}(\llbracket \beta \rrbracket (\sigma)_i) \).

**Proof.** Follows from Lemma C.4.

**Lemma C.8** (\( \beta \) of a set respects renamings of that set).
\[
\llbracket \beta \rrbracket (A_i [a' / a])_i = \llbracket \beta \rrbracket (A_i)_i [a' / a]
\]

**Proof.** Follows from a similar argument to Lemma C.4.

**Lemma C.9** (\( \beta \) commutes with \( \text{dom} \)).
\[
\text{dom}(\llbracket \beta \rrbracket (\sigma)_i) = \llbracket \beta \rrbracket (\text{dom}(\sigma)_i)_i
\]

**Proof.** Follows from a inductive argument on \( \beta \), which is straightforward in every case except \( \beta = \beta' \bowtie \beta'' \) (and \( \cup \)), which is similar. In that case:

\[
\text{dom}(\llbracket \beta' \bowtie \beta'' \rrbracket (\sigma)_i) = \text{dom}(\llbracket \beta' \rrbracket (\sigma)_i)_i \cup \left\{ a \mid a \notin \text{dom}(\llbracket \beta'' \rrbracket (\sigma)_i)_i \right\} \quad \text{By def. of \( \llbracket \beta \rrbracket (\sigma)_i \).}
\]
\[ = \text{dom}(\{J\} (\sigma_i)) \cup \text{dom}(\{J''\} (\sigma_i)) \]
\[ = \{J\} (\text{dom}(\sigma_i)) \cup \{J''\} (\text{dom}(\sigma_i)) \]
\[ = \{J > J''\} (\text{dom}(\sigma_i)) \]

By set arithmetic.

By IH.

By definition of \(>\).

By IH.

By definition of \(\triangleright\).

For lemma C.10, we need to define a notion of the (naive) support of a value.

\[
\text{supp}(\_): \text{Value} \to \text{AtomSet}
\]
\[
\text{supp}(a) \triangleq \{a\}
\]
\[
\text{supp}(\text{inj}_0(v)) \triangleq \text{supp}(v)
\]
\[
\text{supp}(\text{inj}_1(v)) \triangleq \text{supp}(v)
\]
\[
\text{supp}(\text{prod}(v_i)) \triangleq \bigcup_i \text{supp}(v_i)
\]

Lemma C.10 (Support contains all relevant atoms).
\[
\forall v, \tau. \quad \text{fa}(\tau, v) \subseteq \text{supp}(v) \quad \text{and} \quad \text{fb}(\tau, v) \subseteq \text{supp}(v)
\]
\[
\text{fr}(\tau, v) \subseteq \text{supp}(v) \quad \text{and} \quad \text{xa}(\tau, v) \subseteq \text{supp}(v)
\]

Proof. Follows from a simple inductive argument on the size of \(v\).

C.2 Basic \(\alpha\)-equivalence lemmas

Now, we show some basic properties related to \(\alpha\)-equivalence and \(\triangleright\). First, we need to define \(=\). Attempting to compare two values for \(\alpha\)-equivalence is only interesting if they have the same shape.

\[
v = = \text{shape} v' \subseteq \text{Value} \times \text{Value}
\]
\[
\frac{a = = a'}{v = = \text{shape} v'}
\]
\[
\frac{\text{inj}_0(v) = = \text{inj}_0(v')}{v = = \text{shape} v'}
\]
\[
\frac{\text{inj}_1(v) = = \text{inj}_1(v')}{v = = \text{shape} v'}
\]
\[
\frac{\prod_i (v_i) = = \prod_i (v')_i}{v = = \text{shape} v'}
\]

Lemma C.11 (\(=\) is reflexive, preserved under renaming, and implied by \(=\)).
\[
\forall v. \quad v = = v \quad \text{and} \quad \forall a, a'. \quad v = = v [a'/a] \quad \text{and} \quad \forall \tau. \quad v = = v' \quad \tau. \quad v = = v'
\]

Proof. Follows from a straightforward inductive argument on the structure of \(v\).

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

Lemma C.12 (\(\triangleright\) cannot fail, and outputs arbitrary \(\sigma\)'s).
If \(v = = v'\) and \(v : \tau\) and \(A\) is finite, then \(\exists \sigma, \sigma'. v \triangleright v' : \tau \rightarrow \sigma \triangleright \sigma'\) and \(\text{inj}(\sigma)\) and \(\text{inj}(\sigma')\) and \(\text{rng}(\sigma) \# A\) and \(\text{rng}(\sigma') \# A\)

Proof. Follows from a straightforward inductive argument.
Lemma C.13 (≈ is about binders).
If \( v \triangleleft v' : \tau \to \sigma \triangleleft \sigma' \), then \( \text{dom}(\sigma) = \text{fb}(\tau, v) \) and \( \text{dom}(\sigma') = \text{fb}(\tau, v') \)

Proof. By induction on \( \tau \). The interesting case is \( \tau = \text{Prod} \uparrow \beta \)

First:
\[
\forall i. v_i \triangleleft v'_i : \tau_i \to \sigma_i \triangleleft \sigma'_i
\]
\[
\forall i. \text{dom}(\sigma_i) = \text{fb}(\tau_i, v_i)
\]
\[
\sigma = \[\beta_{\text{ex}}\] (\sigma_i)_i
\]
\[
\text{fb}(\tau, v) = \[\beta_{\text{ex}}\] (\text{fb}(\tau_i, v_i))_i
\]
\[
\text{dom}(\sigma) = \text{fb}(\tau, v)
\]

By def. of \( \triangleleft \).

By IH.  
(C.2)

By J-Prod.  
(C.3)

Lemma C.9.

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

Lemma C.14 (Reflexivity of \( \alpha \)-equivalence).
\( v =_R v : \tau \) and \( v =_B v : \tau \) and \( v =_\alpha v : \tau \)

Proof. By a simple inductive argument \( v =_B v : \tau \). We can also use a simple inductive argument to show that \( v =_R v : \tau \).
The \( \text{Ro-Prod} \) case requires another simple inductive argument to show that if \( v \triangleleft v : \tau \to \sigma \triangleleft \sigma' \), then \( \sigma = \sigma' \).

Therefore, \( \[\beta_{\text{in}}\] (\sigma_i)_i = \[\beta_{\text{in}}\] (\sigma'_i)_i \).

Lemma C.15 (Symmetry of \( \alpha \)-equivalence).
\( v =_\alpha v' : \tau \) implies \( v' =_\alpha v : \tau \).

Proof. Follows directly from the symmetric definitions of \( =_\alpha \), \( =_B \), \( =_R \), and \( \triangleleft \).

Lemma C.16 (Transitivity of \( \alpha \)-equivalence).
\( v =_\alpha v' : \tau \) and \( v' =_\alpha v'' : \tau \) implies \( v =_\alpha v'' : \tau \).

Proof. Like Lemma C.14, this follows from a pair of inductive arguments that are trivial in all cases except for \( \text{Ro-Prod} \). In that case:

We have:
\[
\forall i. v_i \triangleleft v'_i : \tau_i \to \sigma_i \triangleleft \sigma'_i \quad \text{and} \quad \forall i. v_i \triangleleft v''_i : \tau_i \to \sigma''_i \triangleleft \sigma''''_i
\]

Define \( \sigma_{\text{join},i} \):
\[
\forall i. \sigma_{\text{join},i} = \sigma_i \circ \text{inv}(\sigma'_i) \circ \sigma''_i
\]

Because \( \forall i. \text{rng}(\sigma'_i) \) are disjoint.

Now:
\[
\forall i. v_i \triangleleft v''_i : \tau_i \to \sigma_{\text{join},i} \triangleleft \sigma''''_i
\]

Because \( \forall i. \sigma_i \) and \( \sigma_{\text{join},i} \)

only differ in their images, which are arbitrary.

And therefore:
\[
\forall i. \[\beta_i\] (\sigma_{\text{join},i})_j (v_i) =_R \[\beta_i\] (\sigma''''_i)_j (v'_i) : \tau
\]

Similarly.
We need an operation on substitutions that can cancel out single-atom renamings on values (see Lemma C.18). We call it “⊗”:

\[ \text{If } a' \notin \text{dom}(\sigma), \sigma \otimes [a'/a] = a \Rightarrow \{ (a_{d\text{-old}} [a'/a], a_{r\text{-old}}) \mid (a_{d\text{-old}}, a_{r\text{-old}}) \in \sigma \} \]

Lemma C.17 (⊗ can be a no-op).
If \( a \notin \text{dom}(\sigma) \), then \( \sigma \otimes [a'/a] = \sigma \).
Proof. Follows from calculation.

Lemma C.18 (⊗ can undo substitutions).
If \( a \in \text{dom}(\sigma) \) and \( a' \neq v \), then \( \sigma \otimes [a'/a](v[a'/a]) = \sigma(v) \).
Proof. By induction on \( v \). The only interesting cases are where \( v \) is an atom:

Suppose \( v = a \):
\[ \langle a, \sigma(a) \rangle \in \sigma \]
\[ \langle a', \sigma(a) \rangle \in \sigma \otimes [a'/a] \]
\[ \sigma \otimes [a'/a](a[a'/a]) = \sigma(a) \]
Or suppose \( v = a_{\text{noop}} \neq a \):
\[ \sigma \otimes [a'/a](a_{\text{noop}}) = \sigma(a_{\text{noop}}) \]
Also:
\[ a_{\text{noop}} \neq a' \]
\[ \sigma \otimes [a'/a](a_{\text{noop}}) = \sigma(a_{\text{noop}}) \]
\[ \sigma \otimes [a'/a](a_{\text{noop}}[a'/a]) = \sigma(a_{\text{noop}}) \]

Lemma C.19 (⊗ adjusts domains).
\( a_d [a'/a] \in \text{dom}(\sigma \otimes [a'/a]) \) iff \( a_d \in \text{dom}(\sigma) \)
Proof. Follows from calculation.

Lemma C.20 (⊗ can distribute over ⪰).
If \( a' \notin \text{dom}(\sigma) \cup \text{dom}(\sigma') \), then \( (\sigma \circ \sigma') \otimes [a'/a] = (\sigma \otimes [a'/a]) \circ (\sigma' \otimes [a'/a]) \)
Proof.
\[
(\sigma \circ \sigma') \otimes [a'/a] = (\sigma \otimes [a'/a]) \cup \left\{ (a_d, a_{r}) \mid a_d \notin \text{dom}(\sigma) \right\}
\]
\[ (\sigma \otimes [a'/a]) \cup \left\{ (a_d, a_{r}) \mid a_d \notin \text{dom}(\sigma \otimes [a'/a]) \right\} \]
By def. of \circ and \otimes.

\[ (\sigma \otimes [a'/a]) \cup \left\{ (a_d, a_{r}) \mid a_d \notin \text{dom}(\sigma \otimes [a'/a]) \right\} \]
By Lemma C.19.

\[ (\sigma \otimes [a'/a]) \cup \left\{ (a_d, a_{r}) \mid a_d \notin \text{dom}(\sigma \otimes [a'/a]) \right\} \]
By def. of \otimes.

\[ (\sigma \otimes [a'/a]) \circ (\sigma' \otimes [a'/a]) \]
By def. of \circ.

\[ (\sigma \otimes [a'/a]) \circ (\sigma' \otimes [a'/a]) \]
By def. of \circ.
Lemma C.21 ($\otimes$ commutes with $\beta$).

$$\llbracket \beta \rrbracket (\sigma_i \otimes [a'/a_i])_i = \llbracket \beta \rrbracket (\sigma_i) \otimes [a'/a]$$

Proof. By induction on $\beta$. All cases are straightforward, except for $\beta = \beta' \cup \beta''$, in which $\llbracket \beta \rrbracket (\sigma_i) = \llbracket \beta' \rrbracket (\sigma_i) \cup \llbracket \beta'' \rrbracket (\sigma_i)_j$, which follows from Lemma C.20.

Lemma C.22 ($\bowtie$ reflects substitutions).

If $v [a'/a] \bowtie v : \tau \rightarrow \sigma \bowtie \sigma'$ and $\text{rng}(\sigma) \neq \text{rng}(\sigma')$, then $\sigma = \sigma' \otimes [a'/a]$

Proof. By induction on $\tau$.

Suppose $\tau = \text{BAtom}$ and $v = a$:

$\sigma = \{a' \rightarrow a_{\text{fresh}}\}$ and $\sigma' = \{a \rightarrow a_{\text{fresh}}\}$

Or suppose $\tau = \text{BAtom}$ and $v = a_v 
eq a$:

$\sigma = \{a_v \rightarrow a_{\text{fresh}}\}$ and $\sigma' = \{a_v \rightarrow a_{\text{fresh}}\}$

Or suppose $\tau = \text{RAtom}$:

$\sigma = \emptyset = \sigma' \otimes [a'/a] = \sigma'$

Or suppose $\tau = \text{Prod}^\text{ex} (\tau_i \downarrow \beta_i)_i$:

$$\forall i. v_i [a'/a] \bowtie v_i : \tau_i \rightarrow \sigma_i \bowtie \sigma'_i$$

$$\llbracket \beta_{\text{ex}} \rrbracket (\sigma_i)_i = \llbracket \beta_{\text{ex}} \rrbracket (\sigma'_i \otimes [a'/a_i])_i = \llbracket \beta_{\text{ex}} \rrbracket (\sigma'_i)_i \otimes [a'/a]$$

$$\sigma = \sigma' \otimes [a'/a]$$

All other cases are trivial.

Lemma C.23 ($\bowtie$ reflects substitution, generalized).

If $v \bowtie v' : \tau \rightarrow \sigma \bowtie \sigma'$, then $v [a'/a] \bowtie v' : \tau \rightarrow \sigma \otimes [a'/a] \bowtie \sigma'$

Proof. By a similar argument to Lemma C.22.

Lemma C.24 (Substitution commutes with free binders).

$$\text{fb}(\tau, \sigma(v)) = \sigma(\text{fb}(\tau, v))$$

Proof. This follows from a straightforward inductive argument.

Lemma C.25 (Substitution commutes with free references).

If $\text{rng}(\sigma) \neq v$ and $\text{inj}(\sigma)$, then $\text{fr}(\tau, \sigma(v)) = \sigma(\text{fr}(\tau, v))$

Proof. This follows from an inductive argument on the size of $v$, which is straightforward in every case except for $v = \text{prod}(v_i)_i$. In that case:

$$\forall i. \text{rng}(\sigma) \neq \text{fr}(\tau_i, v_i) \text{ and } \text{rng}(\sigma) \# \llbracket \beta_i \rrbracket (\text{fb}(\tau_j, v_j))_j$$

Because $\text{rng}(\sigma) \# v$ and by Lemmas C.10 and C.4.

$$(\text{fr}(\tau, \sigma(v)) = \bigcup_{i} (\text{fr}(\tau_i, \sigma(v_i)) \setminus \llbracket \beta_i \rrbracket (\text{fb}(\tau_j, \sigma(v_j)))_j)$$

By def. of fr.
Proof.
Suppose \( a \), Lemma C.28.

\[
\begin{align*}
\text{Suppose } a \text{ and } \tau &= \text{RAtom.} \\
\text{fr}(\text{RAtom}, a) &= \{a\} \\
a \notin \text{fr}(\text{RAtom}, a) &\iff v[a'/a] =^R v: \tau.
\end{align*}
\]

**Lemma C.26 (Substitution commutes with free atoms).**

*If \( \text{rng}(\sigma) \neq v \) and \( \text{inj}(\sigma) \), then \( \text{fa}(\tau, \sigma(v)) = \sigma(\text{fa}(\tau, v)) \)*

**Proof.** This follows directly from Lemmas C.24 and C.25.

**Lemma C.27 (Substitution commutes with exposable atoms).**

*If \( \text{rng}(\sigma) \neq \text{prod} (v_i)_i \) and \( \text{inj}(\sigma) \), then \( \text{xa}(\text{Prod}^{\beta_i} (\tau_i \downarrow v_i)_i, \sigma(\text{prod} (v_i)_i)) = \sigma(\text{xa}(\text{Prod}^{\beta_i} (\tau_i \downarrow v_i)_i, \text{prod} (v_i)_i)) \)*

**Proof.**

\[
\begin{align*}
\text{ xa}(\text{Prod}^{\beta_i} (\tau_i \downarrow v_i)_i, \sigma(\text{prod} (v_i)_i))
&= \bigcup_i \left( \text{fb}(\tau_i, \sigma(v_i)) \setminus \text{fb}(\text{Prod}^{\beta_i} (\tau_i \downarrow v_i)_i, \sigma(\text{prod} (v_i)_i)) \right) \\
&= \sigma \left( \bigcup_i \left( \text{fb}(\tau_i, v_i) \setminus \text{fb}(\text{Prod}^{\beta_i} (\tau_i \downarrow v_i)_i, \text{prod} (v_i)_i) \right) \right) \\
&= \sigma(\text{xa}(\text{Prod}^{\beta_i} (\tau_i \downarrow v_i)_i, \text{prod} (v_i)_i))
\end{align*}
\]

By def. of \( \text{xa} \).

Now, we need to relate free atoms and \( \alpha \)-equivalence. This proof is summarized in section 6.1.2.

**Lemma C.28 (Renaming references respects reference-equivalence iff they are not free).**

*Suppose \( a' \neq v \). Then \( a \notin \text{fr}(\tau, v) \) if and only if \( v[a'/a] =^R v: \tau \).*

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

1. Suppose \( v = a \) and \( \tau = \text{RAtom} \).
   
   \[
   \begin{align*}
   \text{fr}(\text{RAtom}, a) &= \{a\} \\
   a \notin \text{fr}(\text{RAtom}, a) &\iff v[a'/a] =^R a: \text{RAtom}
   \end{align*}
   \]

   By def. of \( \text{fr} \).

   By Ro-RATOM.
2. Suppose $v = a$ and $\tau = \text{BAAtom}$.

\[
\begin{align*}
\text{fr}(\text{BAAtom, } a) &= \varnothing & \text{By def. of fr.} \\
\text{a} \not\in \text{fr}(\text{BAAtom, } a) & \text{iff } v[a'/a] = \text{R } a; \text{BAAtom} & \text{Trivially, by R-BAAtom.}
\end{align*}
\]

3. Suppose $v = \text{prod } (v_i)_i$ and $\tau = \text{Prod}^\text{bij} (\tau_i|\beta_i)_i$.

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

\[
P((\sigma_i)_i, (\sigma'_i)_i) \equiv \forall i. \ v_{\text{sub}}[a'/a] \triangleright v_{\text{sub},i} : \tau_{\text{sub},i} \rightarrow (\sigma_i) \triangleright (\sigma'_i)
\]

and $\text{rng}(\sigma_i), \text{rng}(\sigma'_i) \not\in v[a'/a], v$

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:

\[
P((\sigma_i)_i, (\sigma'_i)_i) 
\]

(C.7)

Fix an $i$. We know:

\[
\sigma_i = \sigma'_i \otimes [a'/a]
\]

By Lemma C.22.

\[
\llbracket \beta_i \rrbracket (\sigma_j)_j = \llbracket \beta_i \rrbracket (\sigma'_j)_j \otimes [a'/a]
\]

By Lemma C.21.

(C.8)

And:

\[
\text{fb}(\tau_{\text{sub},i}, v_{\text{sub},i}) = \text{dom}(\sigma'_i)
\]

By Lemma C.13.

Further assume for the sake of argument:

\[
\text{a} \not\in \llbracket \beta_i \rrbracket (\text{fb}(\tau_{\text{sub},j}, v_{\text{sub},j}))_j
\]

(C.9)

By Lemma C.13.

\[
\text{a} \not\in \llbracket \beta_i \rrbracket (\text{dom}(\sigma'_j))_j
\]

By Lemma C.9.

\[
\text{a} \not\in \text{dom}(\llbracket \beta_i \rrbracket (\sigma'_j)_j)
\]

By C.8 and Lemma C.17.

Trivially.

(C.10)

\[
\text{a} \not\in \triangleright \sigma_i
\]

By IH.

(C.11)

\[
\text{a}' \not\in \text{dom}(\sigma_i)
\]

Because $a' \not\in v$.

(C.11)

\[
\llbracket \beta_i \rrbracket (\sigma_j)_j(v_i)[a'/a] = \llbracket \beta_i \rrbracket (\sigma'_j)_j(v_i)[a'/a]
\]

(C.12)

By C.10.

So:

\[
\forall i. \left( \begin{array}{c}
P((\sigma_j)_j, (\sigma'_j)_j) \\
\text{and a} \not\in \llbracket \beta_i \rrbracket (\text{fb}(\tau_{\text{sub},j}, v_{\text{sub},j}))_j \\
\Rightarrow \llbracket \beta_i \rrbracket (\sigma_j)_j(v_i)[a'/a] = \text{R } \llbracket \beta_i \rrbracket (\sigma'_j)_j(v_i) : \tau_i \\
\text{iff a} \not\in \text{fr}(\tau_i, v_i)
\end{array} \right)
\]

Unassuming C.7 and C.9.

(C.12)

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

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

\[
\exists i. \ a \in \text{fr}(\tau_i, v_i) \text{ and a} \not\in \llbracket \beta_i \rrbracket (\text{fb}(\tau_{\text{sub},j}, v_{\text{sub},j}))_j
\]

By def. of fr.

(C.13)
Suppose a

Lemma C.29

Suppose that

By induction on

Suppose that:

By Lemma C.12.

But:

By C.12 and C.13.

So:

By Rα-Prop and supposition. (C.14)

(b) Suppose that: a \notin \text{fr}(\tau, v)

∀ i. a \notin \text{fr}(\tau_i, v_i) or a \in [\beta_i] ([\text{fb}(\tau_{\text{sub}, j}, v_{\text{sub}, j}))_j

We can have ∀ j. \exists \sigma_j, \sigma'_j.

P((\sigma_j)_j, (\sigma'_j)_j)

By Lemmas C.12 and C.11. (C.16)

Fix an i.

i. Suppose that a \notin [\beta_i] ([\text{fb}(\tau_{\text{sub}, j}, v_{\text{sub}, j}))_j:

\begin{align*}
a \notin \text{fr}(\tau_i, v_i) \\
[\beta_i] (\sigma_j j)(v_i [a'/a]) =_R [\beta_i] (\sigma'_j j)(v_i) : \tau_i
\end{align*}

By definition of fr. (C.17)

By C.12 and C.16. (C.18)

ii. Suppose that a \in [\beta_i] ([\text{fb}(\tau_{\text{sub}, j}, v_{\text{sub}, j}))_j,

In this case, we do not need the IH, because the substitutions generated by \infty already account for the renaming:

\begin{align*}
[\beta_i] (\sigma_j j) = [\beta_i] (\sigma'_j j) \otimes [a'/a] \\
[\beta_i] (\sigma_j j)(v_i [a'/a]) = [\beta_i] (\sigma'_j j)(v_i) \\
[\beta_i] (\sigma_j j)(v_i [a'/a]) =_R [\beta_i] (\sigma'_j j)(v_i) : \tau_i
\end{align*}

By C.16 and Lemmas C.22 and C.21.

By Lemma C.18.

By Lemma C.14.

So:

a \notin \text{fr}(\tau, v) \Rightarrow v [a'/a] =_R v : \tau

By C.18.

And so:

a \in \text{fr}(\tau, v) \iff v [a_v/a] =_{a_v} a_v : \tau

By C.14.

Lemma C.29 (Renaming binders respects binder equivalence iff they are not free).

Suppose a' \neq v. Then a \notin \text{fb}(\tau, v) \iff v [a'/a] =_B v : \tau.

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

1. Suppose v = a_v and \tau = \text{BAtom}:

\begin{align*}
\text{fb(BAtom, v)} = \{a_v\} \\
a \in \{a_v\} \iff v [a'/a] =_B v : \tau
\end{align*}

By def. of fb. 

By Bo-\text{BATOM}.
2. Suppose \( v = a_v \) and \( \tau = \text{RAtom} \): 

\[
\begin{align*}
\text{fb}(&\text{BAtom}, a_v) = \emptyset \\
\alpha \notin \text{fb}(&\text{BAtom}, a_v) \iff v [a'/a] =_{b} a_v : \text{BAtom}
\end{align*}
\]

By def. of fb.

Trivially, by Bo-BAtom.

3. Suppose \( v = \text{prod}(v_i) \) and \( \tau = \text{Prod}^\alpha_{\beta_k} (\tau_i | \beta_i) \): 

\[
\begin{align*}
\alpha \in \text{fb}(&\tau, v) \iff a \in [\beta_{\text{ex}}](\text{fb}(\tau_i, v_i))_i \\
\text{iff } \exists i \in \beta_{\text{ex}}. a \in \text{fb}(\tau_i, v_i) \\
\text{iff } \exists i \in \beta_{\text{ex}}. v_i [a'/a] \neq_b v_i : \tau_i \\
\text{iff } v [a'/a] \neq_b v : \tau
\end{align*}
\]

By def. of fb.

By a straightforward inductive argument.

By Lemma C.30.

Lemma C.30 (Renaming atoms respect \( \alpha \)-equivalence iff they are not free). 

Suppose \( a' \neq v \). Then \( a \notin \text{fa}(\tau, v) \) if and only if \( v [a'/a] =_{\alpha} v : \tau \).

Proof. Follows from Lemmas C.28 and C.29 and \( \alpha \text{Eq} \). □

Now, we prove some collaries of those lemmas.

Lemma C.31 (Non-free atoms can be mass-renamed). 

Suppose \( \text{inj}(\sigma) \) and \( \text{rng}(\sigma) \neq v \) and \( \text{dom}(\sigma) \neq \text{rng}(\sigma) \).

\[
\begin{align*}
\text{dom}(\sigma) \neq \text{fb}(\tau, v) & \Rightarrow \sigma(v) =_{\beta} v : \tau \\
\text{dom}(\sigma) \neq \text{fr}(\tau, v) & \Rightarrow \sigma(v) =_{\alpha} v : \tau \\
\text{dom}(\sigma) \neq \text{fa}(\tau, v) & \Rightarrow \sigma(v) =_{\alpha} v : \tau
\end{align*}
\]

Proof. Follows from Lemmas C.29, C.28, and C.30 and straightforward induction on the size of \( \sigma \). (The disjointness of the domain and range of \( \sigma \) ensures that \( \text{rng}(\sigma) \neq v \) is preserved after individual substitutions.) □

Lemma C.32 (Good renamings preserve binder equivalence).

Suppose \( a' \neq v, v' \). Then \( v =_{\beta} v' : \tau \Rightarrow v [a'/a] =_{b} v' [a'/a] : \tau \)

Proof. By a straightforward inductive argument. □

Lemma C.33 (Good renamings preserve reference equivalence).

Suppose \( a' \neq v, v' \). Then \( v =_{\alpha} v' : \tau \Rightarrow v [a'/a] =_{\beta} v' [a'/a] : \tau \)

Proof. By induction on \( v \). All cases are trivial, except for \( v = \text{prod}(v_i) \):

We have:

\[
\begin{align*}
v =_{\alpha} v' : \tau \\
\forall i, v_i \leadsto v'_i : \tau_i \Rightarrow \sigma_i \bowtie \sigma_i' \text{ and } \forall i, j, \text{ rng}(\sigma_i) \neq v_j, v'_j \\
\forall i, v_i [a'/a] \leadsto v'_i : \tau_i \Rightarrow \sigma_i \bowtie \sigma_i' [a'/a] \\
\forall i, v_i [a'/a] \bowtie v'_i [a'/a] : \tau_i \Rightarrow \sigma_i \bowtie [a'/a] \bowtie \sigma_i' \bowtie [a'/a] \\
\forall i, a' \notin \text{dom}(\beta_i (\sigma_i), \text{ dom}(\beta_i (\sigma_i'))_j)
\end{align*}
\]

By hypothesis.

By inv. of Ro-Prod. (C.19)

By Lemma C.23.

By Lemma C.23, and the symmetry of \( \bowtie \). (C.20)

Because \( a' \neq v, v' \) and by Lemma C.13. (C.21)
Assume:

\( \neg (a \# v \text{ and } a \# v') \)

\( \forall i. a \# \text{rng}(\sigma_i), \text{rng}(\sigma'_i) \)

And:

\( \forall i, j. \text{rng}(\sigma_i \otimes [a'/a]) \# v_j, v'_j \)

Fix an \( i \):

\( \llbracket \beta \rrbracket (\sigma_j \otimes [a'/a])_j = \llbracket \beta \rrbracket (\sigma'_j)_j \otimes [a'/a] \)

And:

\( \llbracket \beta \rrbracket (\sigma'_j \otimes [a'/a])_j = \llbracket \beta_i \rrbracket (\sigma'_j)_j \otimes [a'/a] \)

Also:

\( \llbracket \beta \rrbracket (\sigma_j)_j(v_i) = \llbracket \beta \rrbracket (\sigma'_j)(v'_i) : \tau_i \)

Also:

\( \llbracket \beta \rrbracket (\sigma_j)_j(v_i) [a'/a] = \llbracket \beta \rrbracket (\sigma'_j)(v'_i) [a'/a] : \tau_i \)

Case 1:

\( a \in \text{dom}(\llbracket \beta \rrbracket (\sigma_j)_j) \) and \( a \in \text{dom}(\llbracket \beta \rrbracket (\sigma'_j)_j) \)

\( \llbracket \beta \rrbracket (\sigma_j)_j(v_i [a'/a]) = \llbracket \beta \rrbracket (\sigma_j)(v_i) [a'/a] \)

and \( \llbracket \beta \rrbracket (\sigma'_j)_j(v'_i [a'/a]) = \llbracket \beta \rrbracket (\sigma'_j)(v'_i) [a'/a] \)

\( \llbracket \beta \rrbracket (\sigma_j \otimes [a'/a])_j(v_i [a'/a]) = \llbracket \beta \rrbracket (\sigma_j)(v_i [a'/a]) \)

and \( \llbracket \beta \rrbracket (\sigma'_j \otimes [a'/a])_j(v'_i [a'/a]) = \llbracket \beta \rrbracket (\sigma'_j)(v'_i [a'/a]) \)

\( \llbracket \beta \rrbracket (\sigma_j \otimes [a'/a])_j(v_i [a'/a]) = \llbracket \beta \rrbracket (\sigma'_j \otimes [a'/a])_j(v'_i [a'/a]) : \tau_i \)

Case 2:

\( a \notin \text{dom}(\llbracket \beta \rrbracket (\sigma_j)_j) \) and \( a \notin \text{dom}(\llbracket \beta \rrbracket (\sigma'_j)_j) \)

\( \llbracket \beta \rrbracket (\sigma_j)_j(v_i [a'/a]) = \llbracket \beta \rrbracket (\sigma_j)(v_i) [a'/a] \)

and \( \llbracket \beta \rrbracket (\sigma'_j)_j(v'_i [a'/a]) = \llbracket \beta \rrbracket (\sigma'_j)(v'_i) [a'/a] \)

Likewise.

By Lemma C.21.

(C.24)

By inv. of Ro-Prod.
(C.26)

(C.27)

The other case is trivially proved.

By C.19.

(C.22)

By C.19 and def. of \( \otimes \).

By C.21 and Lemma C.22.

(C.29)

By C.21 and C.22.

(C.28)

By Lemma C.21.

By C.21, C.24, and Lemma C.17.

(C.30)

By C.21 and Lemma C.18.

(WLOG, by symmetry.

\( \forall j. a' \# \text{rng}(\sigma'_j) \).

(C.32)

By C.31 and Lemma C.18.

(C.31)

By C.31 and Lemma C.28.

By C.33 and Lemma C.28.

By C.31.
\[= \beta \rightdownarrow (\sigma')_j (v'_i[a'/a]) \]
\[= ([\beta] (\sigma')_j \otimes [a'/a]) (v'_i[a'/a]) \]
\[= \beta \rightdownarrow (\sigma')_j (v'_i[a'/a]) \]

In all cases:
\[v[a'/a] =_R \ v'[a'/a] : \tau \]

**Lemma C.34** (Good renamings preserve \(\alpha\)-equivalence).

Suppose \(a' \neq v, v'\). Then \(v =_\alpha v' : \tau \Rightarrow v[a'/a] =_\alpha v'[a'/a] : \tau\)

**Proof.** Follows from Lemmas C.32 and C.33.

**Lemma C.35** (Good substitutions preserve \(\alpha\)-equivalence).

Suppose \(\text{rng}(\sigma) \neq v, v'\) and \(\text{dom}(\sigma) \neq \text{rng}(\sigma) \) and \(\text{inj}(\sigma)\).

\[v =_B v' : \tau \Rightarrow \sigma(v) =_B \sigma(v') : \tau \]
\[v =_R v' : \tau \Rightarrow \sigma(v) =_R \sigma(v') : \tau \]
\[v =_\alpha v' : \tau \Rightarrow \sigma(v) =_\alpha \sigma(v') : \tau \]

**Proof.** Follows from Lemmas C.34, C.33, and C.32 and a straightforward induction on the size of \(\sigma\).

**Lemma C.36** (Free atoms are the same for \(\alpha\)-equivalent values).

\[v =_B v' : \tau \Rightarrow \text{fb}(\tau, v) = \text{fb}(\tau, v') \]
\[v =_R v' : \tau \Rightarrow \text{fr}(\tau, v) = \text{fr}(\tau, v') \]
\[v =_\alpha v' : \tau \Rightarrow \text{fa}(\tau, v) = \text{fa}(\tau, v') \]

**Proof.**

\[\forall a, \text{ such that:} \]
\[a \notin \text{fa}(\tau, v) \]
\[v' =_\alpha v : \tau \]

We can choose \(a' \neq v, v'\):

\[v[a'/a] =_\alpha v' : \tau \]

But:

\[v'[a'/a] =_\alpha v : \tau \]

\[a \notin \text{fa}(\tau, v') \]

Therefore, \(\text{fa}(\tau, v) = \text{fa}(\tau, v')\).

The binder and reference versions of that statement follow by similar arguments.

The following are helper lemmas for Lemma C.42.

**Lemma C.37** (Binder-equivalent values don’t need their free binders renamed).

Suppose \(v \bowtie v' : \tau \Rightarrow \sigma \bowtie \sigma'\). If \(v =_B v' : \tau\), then \(\sigma = \sigma'\).
Proof. By an inductive argument on \(v\):

**Case 1.1:**
\(v = a\) and \(\tau = B\text{Atom} \)  
Because \(a =_{B} v' : B\text{Atom}\).

**Case 1.2:**
\(v = a\) and \(\tau = R\text{Atom} \)  
By def. of \(\triangleright \triangleleft\).

**Case 2:**
\(v = \text{prod}(v_i)\) and \(\tau = \text{Prod}_{\beta_i}^{\triangleright}(\tau_i)\)  
By def. of \(\triangleright \triangleleft\).

All other cases are trivial.

**Lemma C.38** (Identical renamings can be removed, retaining reference equality).  
If \(\sigma(v) =_{R} \sigma'(v') : \tau\) and \(\sigma(a) = a' = \sigma'(a)\) and \(a' \# v, v'\),  
then \((\sigma \setminus \{a\})(v) =_{R} (\sigma' \setminus \{a\})(v') : \tau\)

**Proof.**

Case 1: \(a = a'\)

\(a \# v, v'\)

\((\sigma \setminus \{a\})(v) =_{R} \sigma(v) =_{R} \sigma'(v') =_{R} (\sigma' \setminus \{a\})(v') : \tau\)

Case 2: \(a \neq a'\)

\((\sigma \setminus \{a\})(v), \sigma'(v')\)

\(\sigma(v) [a/a'] =_{R} \sigma'(v') [a/a'] : \tau\)

\(\sigma(v) [a/a'] = (\sigma \setminus \{a\})(v)\) and \(\sigma'(v') [a/a'] = (\sigma' \setminus \{a\})(v)\)

\((\sigma \setminus \{a\})(v) =_{R} (\sigma' \setminus \{a\})(v') : \tau\)

**Lemma C.39** (Identical renamings can be removed, retaining binder equality).  
If \(\sigma(v) =_{R} \sigma'(v') : \tau\) and \(\sigma(a) = a' = \sigma'(a)\) and \(a' \# v, v'\),  
then \((\sigma \setminus \{a\})(v) =_{B} (\sigma' \setminus \{a\})(v') : \tau\)

**Proof.** By an argument symmetric to Lemma C.38.
Lemma C.40 (Disjointness of unrelated binders).
Let \( v = \text{prod}(\alpha_i) \) and \( \tau = \text{Prod}^{\beta_{\alpha_i}}(\tau_i, \beta_i) \), and \( \sigma_{\text{all}} = \bigcup \sigma_i' \).

Suppose the following: \( \forall i. \sigma_i' \subseteq \sigma_i \) and \( \text{dom}(\sigma_{\text{all}}) \neq \text{fr}(\tau, v) \) and \( \forall i \neq j. \text{dom}(\sigma_i) \neq \text{dom}(\sigma_j) \) and \( \forall i. \text{fb}(\tau_i, v_i) = \text{dom}(\sigma_i) \).

Then \( \forall i. \text{dom}(\sigma_{\text{all}}) \setminus \text{dom}(\prod \beta_i \, (\sigma_i')) \neq \text{fr}(\tau_i, v_i) \).

Proof.

\[
\forall i. \text{dom}(\prod \beta_i \, (\sigma_i')) = \bigcup_{j \in \beta_i} \text{dom}(\sigma_j')
\]

\[
\forall i. \text{dom}(\sigma_{\text{all}}) \setminus \text{dom}(\prod \beta_i \, (\sigma_i')) \subseteq \bigcup_{j \in \beta_i} \text{dom}(\sigma_j')
\]

\[
\subseteq \bigcup_{j \notin \beta_i} \text{dom}(\sigma_j)
\]

\[
\# \bigcup_{j \in \beta_i} \text{dom}(\sigma_j)
\]

\[
= \prod \beta_i \, (\text{dom}(\sigma_j))_j
\]

By Lemma C.4.

And:

\[
\text{dom}(\sigma_{\text{all}}) \neq \text{fr}(\tau, v)
\]

\[
\text{dom}(\sigma_{\text{all}}) \setminus \text{dom}(\prod \beta_i \, (\sigma_i')) \neq \text{fr}(\tau, v)
\]

By hypothesis.

So:

\[
\forall i. \text{dom}(\sigma_{\text{all}}) \setminus \prod \beta_i \, (\text{dom}(\sigma_j))_j \neq (\text{fr}(\tau, v) \cup \prod \beta_i \, (\text{fb}(\tau_j, v_j))_j)
\]

\[
\forall i. \text{dom}(\sigma_{\text{all}}) \setminus \prod \beta_i \, (\text{dom}(\sigma_j))_j \neq (\text{fr}(\tau, v) \cup \prod \beta_i \, (\text{fb}(\tau_j, v_j))_j)
\]

\[
\forall i. \text{dom}(\sigma_{\text{all}}) \setminus \prod \beta_i \, (\text{dom}(\sigma_j))_j \neq \text{fr}(\tau_i, v_i)
\]

By Lemma C.4. (C.40)

By hypothesis.

By set arithmetic.

Because \( \forall i. \sigma_i' \subseteq \sigma_j \).

By hypothesis.

By set arithmetic.

Because \( \forall i \neq j. \text{dom}(\sigma_i) \neq \text{dom}(\sigma_j) \).

By def. of fr and set arithmetic.

\( \Box \)

Lemma C.41 (\( \bowtie \) makes pairs of binder-disjoint values binder-equal).
If \( \bowtie_{\text{bndrs-disj}} \, u : \tau \) and \( \bowtie_{\text{bndrs-disj}} \, u' : \tau \) then

\[
v \bowtie u : \tau \rightarrow \sigma \bowtie \sigma' : \tau \Rightarrow \sigma(v) =_{b} \sigma'(v') : \tau
\]

Proof. By induction on the size of \( v \). All cases are trivial, except for \( v = \text{prod} \, (\alpha_i) \) and \( \tau = \text{Prod}^{\beta_{\alpha_i}}(\tau_i, \beta_i) \). In that case, it follows from Lemma C.6. \( \Box \)

The purpose of the following lemma (discussed in section 6.1.1) is to show that any pair of values that are \( \alpha \)-equivalent and sufficiently-disjoint are suitable for unpacking into the environment with the free names \( A \). By this we mean that a single pair of substitutions can make their subterms \( \alpha \)-equivalent, not one pair per pair of subterms.

Lemma C.42 (Sufficiently-disjoint \( \alpha \)-equivalent values can be opened).
Let \( \tau_{\text{obj}} = \text{Prod}^{\beta_{\alpha_i}}(\tau_i, \beta_i) \) and \( v = \text{prod} \, (\alpha_i) \) and \( v' = \text{prod} \, (\alpha_i') \).

Suppose \( \text{fa}(\tau_{\text{obj}}, v) \) and \( \text{fa}(\tau_{\text{obj}}, v') \) and \( \text{fa}(\tau_{\text{obj}}, v \bowtie v') \) and \( A \) is finite.

If \( v =_{\alpha} v' : \tau_{\text{obj}} \), then \( : \sigma \bowtie \sigma' : \text{dom}(\sigma) \subseteq \text{xa}(\tau_{\text{obj}}, \sigma) \) and \( \text{dom}(\sigma') \subseteq \text{xa}(\tau_{\text{obj}}, \sigma') \) and \( \forall i. \sigma(v_i) =_{\alpha} \sigma'(v'_i) : \tau_i \) and \( \text{rng}(\sigma) \neq A \) and \( \text{rng}(\sigma') \neq A \) and \( \text{inj}(\sigma) \) and \( \text{inj}(\sigma') \).
Proof.

\( \forall i. v_i \vdash v'_i : \tau_i \rightarrow \sigma_{\text{orig},i} \vdash \sigma'_{\text{orig},i} \)

\( \forall i, j. \text{rng}(\sigma_{\text{orig},i}) \cup \text{rng}(\sigma'_{\text{orig},i}) \neq v_j, v'_j \)

\( \forall i. [\beta_i] (\sigma_{\text{orig},j}) (v_i) = R [\beta_i] (\sigma'_{\text{orig},j}) (v'_i) : \tau_i \)

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

\( \text{dom}(\sigma_{\text{disj}}) = \bigcup \text{rng}(\sigma_{\text{orig},i}) \cup \text{rng}(\sigma'_{\text{orig},i}) \)

\( \text{inj}(\sigma_{\text{disj}}) \) and \( \forall i. \text{rng}(\sigma_{\text{disj}}) \neq v_i, v'_i \) and \( \text{rng}(\sigma_{\text{disj}}) \neq A \)

To keep exported binders fixed, define:

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

Define:

\( \sigma_{\text{local},i} \triangleq \left\{ \begin{array}{ll}
\varnothing & i \notin \beta_{\text{ex}} \\
\sigma_i & i \in \beta_{\text{ex}}
\end{array} \right. \) and \( \sigma'_{\text{local},i} \triangleq \left\{ \begin{array}{ll}
\varnothing & i \notin \beta_{\text{ex}} \\
\sigma'_i & i \in \beta_{\text{ex}}
\end{array} \right. \)

Now:

\( \forall i, j. \text{rng}(\sigma_{\text{local},i}) \cup \text{rng}(\sigma'_{\text{local},i}) \neq v_j, v'_j, A \)

For convenience, define:

\( \sigma_{\text{imp},i} \triangleq \llbracket \beta_i \rrbracket (\sigma_{\text{local},i})_j \) and \( \sigma'_{\text{imp},i} \triangleq \llbracket \beta_i \rrbracket (\sigma'_{\text{local},i})_j \)

\( \forall j \notin \beta_{\text{ex}} \cdot \sigma_{\text{imp},i}(v_i) = R \sigma'_{\text{imp},i}(v'_i) : \tau_i \)

\( \text{dom}(\sigma_{\text{local},i}) \subseteq \text{fb}(\tau_i, v_i) \) and \( \text{dom}(\sigma'_{\text{local},i}) \subseteq \text{fb}(\tau_i, v'_i) \)

But:

\( v =_B v' : \tau_{\text{obj}} \)

\( \forall i \notin \beta_{\text{ex}} \cdot v_i =_B v'_i : \tau_i \)

\( \forall i \in \beta_{\text{ex}} \cdot \sigma_i = \sigma'_i \)

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

\( \forall i. \sigma_{\text{imp},i}(v_i) = R \sigma'_{\text{imp},i}(v'_i) : \tau_i \)

Now:

\( \forall i, j \in \beta_{\text{ex}} \cdot i \neq j \Rightarrow \text{dom}(\sigma_{\text{local},i}) \neq \text{dom}(\sigma_{\text{local},j}) \)

And:

\( \forall i \neq j. \text{dom}(\sigma_{\text{local},i}) \neq \text{dom}(\sigma_{\text{local},j}) \)

So, define a single substitution:

\( \sigma \triangleq \bigcup_i \sigma_{\text{local},i} \)

\( \text{dom}(\sigma) \neq \text{rng}(\sigma) \)

But:

\( \text{dom}(\sigma) \neq \text{fb}(\tau_{\text{obj}}, v) \)

By def. of \( \tau_{\text{obj}} \) and Lemma C.47.

So it doesn’t matter whether a subterm is exported:
\[
\begin{align*}
\text{dom}(\sigma) & \subseteq \text{xa}(\tau_{\text{obj}}, v) \\
\text{dom}(\sigma) & \# \text{fa}(\tau_{\text{obj}}, v) \\
\text{dom}(\sigma) & \# \text{fr}(\tau_{\text{obj}}, v) \\
\forall i. \ \text{dom}(\sigma) \setminus \text{dom}(\sigma_{\text{imp}}, i) & \# \text{fr}(\tau_i, v_i) \\
\text{And:} & \\
\forall i. \ \text{dom}(\sigma) \setminus \text{dom}(\sigma_{\text{imp}, i}) & \# \bigcup_j \text{rng}(\sigma_i) \\
\text{But:} & \\
\forall i. \ \text{fr}(\tau_i, \sigma_{\text{imp}, i}(v_i)) & = \sigma_{\text{imp}, i}(\text{fr}(\tau_i, v_i)) \\
& \subseteq (\text{fr}(\tau_i, v_i) \setminus \text{dom}(\sigma_{\text{imp}, i})) \cup \text{rng}(\sigma_{\text{imp}, i}) \\
& \subseteq \text{fr}(\tau_i, v_i) \cup \text{rng}(\sigma_{\text{imp}, i}) \\
& \subseteq \text{fr}(\tau_i, v_i) \cup \bigcup_j \text{rng}(\sigma_j) \\
& \# \text{dom}(\sigma) \setminus \text{dom}(\sigma_{\text{imp}, i}) \\
\text{fr}(\tau_i, \sigma_{\text{imp}, i}(v_i)) & \# \text{dom}(\sigma \setminus \sigma_{\text{imp}, i}) \\
\sigma_{\text{imp}, i} & \subseteq \sigma \\
\forall i. \ \sigma_{\text{imp}, i}(v_i) & = R (\sigma \setminus \sigma_{\text{imp}, i})(\sigma_{\text{imp}, i}(v_i)) \cdot \tau_i \\
\forall i. \ \tau_i & = \sigma(v_i) \\
\text{And:} & \\
\sigma' & \triangleq \bigcup_i \sigma'_{\text{local}, i} \quad \text{and} \quad \forall i. \ \sigma'_{\text{imp}, i}(v'_i) = R \sigma'(v'_i) \cdot \tau_i \\
\forall i. \ \sigma(v_i) & = R \sigma'(v'_i) : \tau_i \\
\text{Also:} & \\
\forall i. \ \beta_{\text{ex}} \cdot \text{dom}(\sigma) & \# \text{fb}(\tau_i, v_i) \quad \text{and} \quad \text{dom}(\sigma') \# \text{fb}(\tau_i, v'_i) \\
\forall i. \ \beta_{\text{ex}} \cdot \sigma(v_i) & = B \sigma'(v'_i) : \tau_i \\
\text{And:} & \\
\forall i. \ \beta_{\text{ex}} \cdot \sigma_{\text{orig}, i}(v_i) & = B \sigma'_{\text{orig}, i}(v'_i) : \tau_i \\
\forall i. \ \beta_{\text{ex}} \cdot \sigma_i(v_i) & = B \sigma'_i(v'_i) : \tau_i \\
\forall i. \ \beta_{\text{ex}} \cdot \sigma_{\text{local}, i}(v_i) & = B \sigma'_{\text{local}, i}(v'_i) : \tau_i \\
\forall i. \ \beta_{\text{ex}} \cdot \sigma(v_i) & = B \sigma'(v'_i) : \tau_i \\
\text{Finally:} & \\
\forall i. \ \sigma(v_i) & = \sigma'(v'_i) : \tau_i \\
\text{Furthermore, we have:} & \\
\text{dom}(\sigma) & \subseteq \text{xa}(\tau_{\text{obj}}, v) \quad \text{and} \quad \text{dom}(\sigma') \subseteq \text{xa}(\tau_{\text{obj}}, v') \\
\text{rng}(\sigma) & \# A \quad \text{and} \quad \text{rng}(\sigma') \# A \\
\text{inj}(\sigma) \quad \text{and} \quad \text{inj}(\sigma') \\
\end{align*}
\]

By def. of xa. \hspace{1cm} (C.58)
By def. of \(\vdash\) suff-disj. \hspace{1cm} (C.59)
By def. of fa.
By C.54 on Lemma C.40.
By C.47 and C.50 and Lemma C.10.
By C.47 and Lemma C.25.
By set arithmetic.
By set arithmetic.
By Lemma C.3.
By C.59 and C.60.
By set arithmetic.
By C.61 and C.56 on Lemma C.31.
By C.56.
By a similar argument.
By C.57 and def. of fb and Lemma C.3.
By C.47 and C.56 on Lemma C.31.
By def. of \(\vdash\) suff-disj and C.41 on Lemma C.41.
By C.44 on Lemma C.31.
Because
\[\forall i. \beta_{\text{ex}} \cdot \sigma_{\text{local}, i} = \sigma_i \quad \text{and} \quad \sigma'_{\text{local}, i} = \sigma'_i,\]
By C.44 on Lemma C.31.
By C.63 and C.65.
By C.58.
By C.44.
By C.41 and C.44.
Lemma C.43 (New, disjoint, binders can be generated).
Given \( \text{prod} (v_1)_i : \text{Prod} (\tau_i \downarrow \beta_i)_i \) and finite \( A : \forall \in \).
\( \exists \sigma_i, \) such that \( v_i \gg v_i : \tau_i \rightarrow \sigma_i \gg \sigma_i \) 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 \( \forall j. \text{rng}(\sigma_i) \# \text{dom}(\sigma_j) \).

Proof. Follows from Lemmas C.11, C.13, C.12, and C.37 by straightforward induction on \( i \).

Lemma C.44 (Products with binder-equivalent components are binder-equivalent).
If \( \forall i. v_i =_B v'_i : \tau_i \), then \( \text{prod} (v_1)_i =_B \text{prod} (v'_1)_i : \text{Prod}^{\beta_i} (\tau_i \downarrow \beta_i)_i \).

Proof. Follows from \( B_0 \)-Prod.

Lemma C.45 (Products with \( \alpha \)-equivalent components are \( \alpha \)-equivalent).
If \( \forall i. v_i =_\alpha v'_i : \tau_i \), then \( \text{prod} (v_1)_i =_\alpha \text{prod} (v'_1)_i : \text{Prod}^{\beta_i} (\tau_i \downarrow \beta_i)_i \).

Proof.
First:
\( \forall i. v_i =_\alpha v'_i : \tau_i \)
\( \exists \sigma_i : \)
\( \forall i. v_i \gg v'_i : \tau_i \rightarrow \sigma_i \gg \sigma_i \)
\( \text{inj}(\sigma_i) \) and \( \text{rng}(\sigma_i) \# v_i \) and \( \forall j. \text{rng}(\sigma_i) \# \text{dom}(\sigma_j) \)
Also:
\( \forall i \neq j. \text{rng}(\sigma_i) \# \text{rng}(\sigma_j) \)
Likewise.
By hypothesis.  \( \text{(C.68)} \)
By Lemma C.43. \( \text{(C.69)} \)
By Lemma C.43.
By Lemma C.44.

Lemma C.46 (Substitutions don’t affect \( \text{FAULT} \)’s uniqueness).
\( \sigma (w) = \text{FAULT} \) iff \( w = \text{FAULT} \)

Proof. Follows directly from the definition of \( \sigma (w) \).

Lemma C.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 \( \text{rng}(\text{inv}(\sigma)) \# \sigma(v) \).

Proof. Follows from straightforward set operations.

Lemma C.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 \( A \vdash \text{suff-disj} \sigma (v) : \tau \)

Proof. This follows by a straightforward argument from Lemmas C.1 and C.26.

Lemma C.49 (Good substitutions preserve \( \vdash \text{suff-disj} \)).
If \( \text{inj}(\sigma) \) and \( \text{rng}(\sigma) \# A, v \) and \( A \vdash \text{suff-disj} \sigma (v) : \tau \), then \( \sigma(A) \vdash \text{suff-disj} \sigma (v) : \tau \).
Proof. First, by Lemma C.24 and inj(σ), we know \(\forall \tau, v, v'. \, fb(\tau, v) \# fb(\tau, v') \Rightarrow fb(\tau, \sigma(v)) \# fb(\tau, \sigma(v'))\). And, by Lemma C.27 and inj(σ) and rng(σ) \# v, A, we know \(\forall \tau, v, v'. \, xa(\tau, v) \# A \Rightarrow xa(\tau, \sigma(v)) \# \sigma(A)\).

Then, the proof proceeds by a pair of straightforward inductive arguments, one for \(\vdash \text{bndrs-disj}\), and one for \(\vdash \text{suff-disj}\) itself.

**Lemma C.50** (Range update can be expressed in terms of composition).
If \(\text{rng}(\sigma) \# v\), then for all \(\sigma\text{avoid}\), we have \(\{\langle a, \sigma\text{avoid}(a') \rangle \mid \langle a, a' \rangle \in \sigma\}(v) = \sigma\text{avoid}(\sigma(v))\)

Proof. Follows from a straightforward inductive argument on the size of \(v\).

**Lemma C.51** (Simultaneous range updates respect \(\alpha\)-equivalence).
If \(\text{rng}(\sigma) \# v\) and \(\text{rng}(\sigma') \# v'\) and inj(\(\sigma\text{avoid}\)) and \(\text{rng}(\sigma\text{avoid}) \# \sigma(v)\), \(\sigma'(v')\) and dom(\(\sigma\text{avoid}\)) \# rng(\(\sigma\text{avoid}\)), then
\[
\{\langle a, \sigma\text{avoid}(a') \rangle \mid \langle a, a' \rangle \in \sigma\}(v) = \sigma = \alpha \{\langle a, \sigma\text{avoid}(a') \rangle \mid \langle a, a' \rangle \in \sigma'\}(v') \text{ iff } \sigma(v) = \alpha \sigma'(v') : \tau
\]

Proof. Follows from Lemmas C.50, C.35, and C.47

### C.3 Execution soundness itself

**Theorem C.1** (Determinism up to \(\alpha\)-equivalence, termination-insensitive version).

If \(\tau = \text{typeof}(\Gamma, e)\)
and \(\rho = \alpha \rho' : \Gamma\)
and \(\Gamma \vdash_{\text{exe}} \langle e, \rho \rangle \kern k \Rightarrow w\)
and \(\Gamma \vdash_{\text{exe}} \langle e, \rho' \rangle \kern k' \Rightarrow w'\)
then \(w = \alpha w' : \tau\)

Proof. By induction on \(k\), and case analysis on \(e\) in:

- **Case E-FRESH-*:**

  First:
  \[
  \Gamma \vdash_{\text{exe}} \langle (\text{fresh } x \text{ in } e), \rho \rangle \overset{k+1}{\Rightarrow} w_{\text{out}}
  \]
  \[
  \Gamma \vdash_{\text{exe}} \langle (\text{fresh } x \text{ in } e), \rho' \rangle \overset{k'+1}{\Rightarrow} w'_{\text{out}}
  \]
  Define:
  \[\Gamma_{\text{ih}} \triangleq \Gamma, x : \text{BAtom}\]
  We know:
  \[\exists a. \, \Gamma_{\text{ih}} \vdash_{\text{exe}} \langle e, \rho [x \rightarrow a] \rangle \kern k \Rightarrow w\]

  By hypothesis. \(\text{C.71}\)

  By inv. of E-FRESH-*. \(\text{C.72}\)
\[ \exists a'. \text{IH} \vdash \text{exec } \{ e, \rho' [x \rightarrow a'] \} \xrightarrow{t} w' \]

inj(\sigma) \text{ and } inj(\sigma')

rng(\sigma) \# \rho, w, a

rng(\sigma') \# \rho', w', a'

\[ \sigma(\rho) =_{\alpha} \sigma'(\rho') : \Gamma \]

Choose:

\[ \alpha_{1r} \# \sigma, \sigma', w, w', \rho, \rho', a, a' \]

Define:

\[ \sigma_{\text{IH}} \triangleq \{ (\sigma(a), a_1) \} \circ \sigma \]

\[ \sigma'_{\text{IH}} \triangleq \{ (\sigma'(a'), a_1) \} \circ \sigma' \]

\[ \sigma_{\text{IH}}(a) = \sigma'_{\text{IH}}(a') \]

And:

\[
\begin{align*}
\alpha & \notin \text{fa}_{\text{env}}(\Gamma, \rho) \text{ and } \alpha' \notin \text{fa}_{\text{env}}(\Gamma, \rho') \\
\sigma(\alpha) & \notin \text{fa}_{\text{env}}(\Gamma, \sigma(\rho)) \\
\sigma'(\alpha') & \notin \text{fa}_{\text{env}}(\Gamma, \sigma(\rho')) \\
\sigma_{\text{IH}}(\rho) & =_{\alpha} \sigma'_{\text{IH}}(\rho') : \Gamma \\
\sigma_{\text{IH}}(\rho[x \rightarrow a]) & =_{\alpha} \sigma'_{\text{IH}}(\rho'[x \rightarrow a']: \Gamma_{\text{IH}})
\end{align*}
\]

And:

\[ \text{inj}(\sigma_{\text{IH}}) \text{ and } \text{inj}(\sigma'_{\text{IH}}) \]

\[ \text{rng}(\sigma_{\text{IH}}) \# \rho[x \rightarrow a], w \]

\[ \text{rng}(\sigma'_{\text{IH}}) \# \rho'[x \rightarrow a'], w' \]

So:

\[
\begin{align*}
\sigma_{\text{IH}}(w) & =_{\alpha} \sigma'_{\text{IH}}(w') : \tau
\end{align*}
\]

By straightforward substitution.

By inv. of E-Fresh-\(*\).

By C.74 and Lemma C.26.

By C.74 on Lemma C.30.

By hypothesis.

By inv. of E-Fresh-\(*\).

By C.78 on Lemma C.30.

By C.74 and C.80 on IH.

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

1. Suppose that \(w = \text{FAULT or } w' = \text{FAULT}:

\[
\begin{align*}
w & = w' = \text{FAULT} \\
w_{\text{out}} & = \text{FAULT and } w'_{\text{out}} = \text{FAULT}
\end{align*}
\]

By C.82 on Lemma C.46.

By C.72 and C.73 and E-Fresh-\(*\).

2. Suppose that \(w \neq \text{FAULT and } w' \neq \text{FAULT}:

\[
\begin{align*}
\text{fa}(\tau, \sigma_{\text{IH}}(w)) & = \text{fa}(\tau, \sigma'_{\text{IH}}(w')) \\
\sigma_{\text{IH}}(\text{fa}(\tau, w)) & = \sigma'_{\text{IH}}(\text{fa}(\tau, w')) \\
\sigma_{\text{IH}}(a) & \notin \text{fa}(\tau, \sigma_{\text{IH}}(w)) \\
\text{iff } \sigma'_{\text{IH}}(a') & \notin \text{fa}(\tau, \sigma'_{\text{IH}}(w')) \\
a & \notin \text{fa}(\tau, w) \text{ iff } a' \notin \text{fa}(\tau, w')
\end{align*}
\]

So, suppose that both names are not free:

\[
\begin{align*}
a & \notin \text{fa}(\tau, w) \text{ and } a' \notin \text{fa}(\tau, w') \\
w_{\text{out}} & = w \text{ and } w'_{\text{out}} = w'
\end{align*}
\]

By C.72 and C.73 and E-Fresh-\(*\).

78
And also:
\[ \sigma'(a) \notin \text{fa}(\tau, \sigma'(w)) \]
and \[ \sigma'(a') \notin \text{fa}(\tau, \sigma'(w')) \]
\[ \sigma(w) = \alpha \sigma'(w') : \tau \]
\[ \sigma(w) = \alpha \sigma'(w') : \tau \]
\[ \sigma'(w) = \alpha \sigma'(w') : \tau \]

(b) Alternatively, suppose \( a \in \text{fa}(\tau, w) \) and \( a' \in \text{fa}(\tau, w') \):
\[ \sigma'(w) = \alpha \sigma'(w') : \tau \]
\[ \sigma'(w) = \alpha \sigma'(w') : \tau \]
\[ \sigma(w) = \alpha \sigma'(w') : \tau \]

Case E-Open++: Let \( v_{\text{obj}} \triangleq \text{prod} (v_1)_1 \) and \( v'_{\text{obj}} \triangleq \text{prod} (v'_1)_1 \) and \( e_{\text{all}} \triangleq (\text{open} x_{\text{obj}} ((x_i)_1) e) \).

First:
\[ \Gamma \vdash \text{exe} \langle e_{\text{all}}, \rho \rangle \xrightarrow{k+1} w_{\text{out}} \]
and \[ \Gamma \vdash \text{exe} \langle e_{\text{all}}, \rho' \rangle \xrightarrow{k'+1} w'_{\text{out}} \]
Define:
\[ \Gamma_{\text{IH}} \triangleq \Gamma, (x_i : \tau_i)_1 \]
Now:
\[ \forall i. \exists v_i. \Gamma_{\text{IH}} \vdash \text{exe} \langle e, \rho [x_i \rightarrow v_i]_1 \rangle \xrightarrow{k} w \]
\[ \text{fa}_{\text{env}}(\Gamma, \rho) \vdash \text{suff-disj} v_{\text{obj}} : \tau_{\text{obj}} \]
\[ \forall i. \exists v_i. \Gamma_{\text{IH}} \vdash \text{exe} \langle e, \rho' [x_i \rightarrow v'_i]_1 \rangle \xrightarrow{k'} w' \]
\[ \text{fa}_{\text{env}}(\Gamma, \rho') \vdash \text{suff-disj} v'_{\text{obj}} : \tau_{\text{obj}} \]
\[ v_{\text{obj}} \triangleq \text{prod} (v_1)_1 = \alpha \rho (x_{\text{obj}}) : \tau_{\text{obj}} \]
\[ v'_{\text{obj}} \triangleq \text{prod} (v'_1)_1 = \alpha \rho' (x_{\text{obj}}) : \tau_{\text{obj}} \]
\[ \sigma(\rho) = \alpha \sigma'(\rho') : \Gamma \]
And:
\[ \sigma(v_{\text{obj}}) = \alpha \sigma(\rho (x_{\text{obj}})) : \tau_{\text{obj}} \]
\[ = \alpha \sigma'(\rho' (x_{\text{obj}})) : \tau_{\text{obj}} \]
\[ = \alpha \sigma'(v'_{\text{obj}}) : \tau_{\text{obj}} \]
\[ \sigma(v_{\text{obj}}) = \alpha \sigma'(v'_{\text{obj}}) : \tau_{\text{obj}} \]
And:
\[ \sigma(\text{fa}_{\text{env}}(\Gamma, \rho)) \vdash \text{suff-disj} \sigma(v_{\text{obj}}) : \tau_{\text{obj}} \]
\[ \sigma'(\text{fa}_{\text{env}}(\Gamma, \rho')) \vdash \text{suff-disj} \sigma'(v'_{\text{obj}}) : \tau_{\text{obj}} \]
\[ \sigma(\text{fa}(\tau_{\text{obj}}, v_{\text{obj}})) \vdash \text{suff-disj} \sigma(\tau_{\text{obj}}) : \tau_{\text{obj}} \]
\[ \sigma'(\text{fa}(\tau_{\text{obj}}, v'_{\text{obj}})) \vdash \text{suff-disj} \sigma'(\tau_{\text{obj}}) : \tau_{\text{obj}} \]
\[ \exists \sigma_{\text{op}}, \sigma'_{\text{op}}. \]
\[ \text{dom} (\sigma_{\text{op}}) \subseteq \text{xa}(\tau_{\text{obj}}, \sigma(\tau_{\text{obj}})) \]
\[ \text{dom} (\sigma'_{\text{op}}) \subseteq \text{xa}(\tau_{\text{obj}}, \sigma'(\tau_{\text{obj}})) \]

By C.86 and C.74 and Lemma C.26. (C.88)
By C.82 and C.76 on Lemma C.30. (C.90)
By C.76 on Lemma C.51. (C.96)
By C.87. (C.99)

By C.72 and C.73 and E-Fresh++. (C.90)

By inv. of E-Open++ on C.90. (C.91)
Likewise. (C.92)
Likewise. (C.93)
Likewise. (C.94)
Likewise. (C.95)
By hypothesis. (C.96)

By C.95 and Lemma C.35. (C.97)
By C.96. (C.98)
By C.95 and Lemma C.35. (C.99)
By C.97–C.98 and Lemma C.16. (C.100)
By C.92 and C.94 and Lemma C.49. (C.101)
By C.97 and C.98 and Lemma C.36. (C.101)
By C.92 on Lemma C.42. (C.102)
Such that:
\[ \forall i. \sigma_{op}(\sigma(v_i)) = \alpha, \sigma'_{op}(\sigma'(v'_i)) : \tau \]
\[ \text{rng}(\sigma_{op}), \text{rng}(\sigma'_{op}) \neq \sigma, \sigma', \rho, \rho', w, w', v_{obj}, v'_{obj} \]
\[ \text{inj}(\sigma_{op}) \text{ and inj}(\sigma'_{op}) \]
And:
\[ \text{dom}(\sigma_{op}), \text{dom}(\sigma'_{op}) \neq \text{rng}(\sigma_{op}), \text{rng}(\sigma'_{op}) \]
Define:
\[ \sigma_{IH} \triangleq \sigma_{op} \circ \sigma \text{ and } \sigma'_{IH} \triangleq \sigma'_{op} \circ \sigma' \]
Now:
\[ \text{xa}(\tau_{obj}, \sigma(v_{obj})) \neq \text{fa}_{env}(\Gamma, \sigma(\rho)) \]
\[ \text{xa}(\tau_{obj}, \sigma'(v'_{obj})) \neq \text{fa}_{env}(\Gamma, \sigma'(\rho')) \]
\[ \sigma_{IH}(\rho) = \alpha, \sigma'_{IH}(\rho') : \Gamma \]
\[ \sigma_{IH}(\rho [x_i \rightarrow v_i]) = \alpha, \sigma'_{IH}(\rho' [x_i \rightarrow v'_i]) : \Gamma_{IH} \]
And:
\[ \text{inj}(\sigma_{IH}) \text{ and inj}(\sigma'_{IH}) \]
And:
\[ \text{rng}(\sigma_{IH}) \neq \rho [x_i \rightarrow v_i] \]
\[ \text{rng}(\sigma'_{IH}) \neq \rho' [x_i \rightarrow v'_i] \]
So:
\[ \sigma_{IH}(w) = \alpha, \sigma'_{IH}(w') : \tau \]
Case 1:
\[ w = \text{FAULT or } w' = \text{FAULT} \]
\[ w = w' = \text{FAULT} \]
\[ w_{out} = w'_{out} = \text{FAULT} \]
Case 2:
\[ w \neq \text{FAULT and } w' \neq \text{FAULT} \]
We know:
\[ \sigma_{IH}(v_{obj}) = \alpha, \sigma_{IH}(v'_{obj}) : \tau_{obj} \]
\[ \text{xa}(\tau_{obj}, \sigma_{IH}(v_{obj})) = \text{xa}(\tau_{obj}, \sigma'_{IH}(v'_{obj})) \]
\[ \sigma_{IH}(\text{xa}(\tau_{obj}, v_{obj})) = \sigma'_{IH}(\text{xa}(\tau_{obj}, v'_{obj})) \]
And:
\[ \text{fa}(\tau, \sigma_{IH}(w)) = \text{fa}(\tau, \sigma'_{IH}(w')) \]
\[ \sigma_{IH}(\text{fa}(\tau, w)) = \sigma'_{IH}(\text{fa}(\tau, w')) \]
Likewise. (C.102)
Likewise. (C.103)
Likewise. (C.104)
By C.101 and C.103 and Lemma C.10. (C.105)
By C.100 and def. of \text{IH}^\ast \text{suff-disj} and Lemma C.26.
By C.102. (C.106)
Because \text{inj}(\sigma) \text{ and inj}(\sigma') \text{ and by C.103 and C.104.} (C.107)
Because \text{rng}(\sigma) \neq \rho \text{ and } \text{rng}(\sigma') \neq \rho' \text{ and by C.103.} (C.108)
By C.107 and C.108 on IH. (C.109)
By C.109 on Lemma C.46. (C.110)
By C.91 and C.93 and E-Open\#. (C.111)
By C.102 on Lemma C.45.
By def. of \text{xa} and C.102 and Lemma C.36.
By C.103 and C.108 and Lemma C.27. (C.112)
By C.109 on Lemma C.36. (C.113)
By C.108 on Lemma C.26. (C.114)
\[ \sigma_{\text{IH}}(xa(\tau_{\text{obj}}, v_{\text{obj}})) \# \sigma_{\text{IH}}(fa(\tau, w)) \]
\[
\text{iff } \sigma'_{\text{IH}}(xa(\tau_{\text{obj}}, v'_{\text{obj}})) \# \sigma'_{\text{IH}}(fa(\tau, w'))
\]
\[ xa(\tau_{\text{obj}}, v_{\text{obj}}) \# fa(\tau, w) \]
\[
\text{iff } xa(\tau_{\text{obj}}, v'_{\text{obj}}) \# fa(\tau, w')
\]

So, case 2.1:

\[ xa(\tau_{\text{obj}}, v_{\text{obj}}) \# fa(\tau, w) \]
\[
\text{and } xa(\tau_{\text{obj}}, v'_{\text{obj}}) \# fa(\tau, w')
\]
\[ w_{\text{out}} = w \text{ and } w'_{\text{out}} = w' \]
And also:
\[ \sigma(xa(\tau_{\text{obj}}, v_{\text{obj}})) \# \sigma(fa(\tau, w)) \]
\[ \sigma'(xa(\tau_{\text{obj}}, v'_{\text{obj}})) \# \sigma'(fa(\tau, w')) \]
\[ \text{dom}(\sigma_{\text{op}}) \# \sigma(fa(\tau, w')) \]
\[ \text{dom}(\sigma'_{\text{op}}) \# \sigma'(fa(\tau, w')) \]
\[ \sigma_{\text{IH}}(w) =_{\alpha} \sigma(w) : \tau \]
\[ \sigma'(w') =_{\alpha} \sigma'(w') : \tau \]
\[ \sigma(w) =_{\alpha} \sigma'(w') : \tau \]
\[ \sigma'(w_{\text{out}}) =_{\alpha} \sigma'(w'_{\text{out}}) : \tau \]

Or, case 2.2:
\[ w_{\text{out}} = \text{FAULT and } w'_{\text{out}} = \text{FAULT} \]

• Case \text{E-CALL}:

Define:
\[ \Gamma_{\text{IH}} \triangleq (x_i \tau_i)_i \]

First:
\[ \Gamma_{\text{IH}} \vdash \text{exe } e, [x_i \rightarrow \rho(x_{\text{actual},i})]_i \Rightarrow w \]

And:
\[ \Gamma_{\text{IH}} \vdash \text{exe } e, [x_i \rightarrow \rho'(x_{\text{actual},i})]_i \Rightarrow w' \]

But:
\[ \sigma([x_i \rightarrow \rho(x_{\text{actual},i})]_i) =_{\alpha} \sigma'([x_i \rightarrow \rho'(x_{\text{actual},i})]_i) : \Gamma_{\text{IH}} \]
\[ \sigma(w) =_{\alpha} \sigma'(w') : \tau \]

• Case \text{E-LET-*}: Let \( e_{\text{all}} \triangleq (\text{let } x \text{ where } C \text{ be } e_{\text{val}} \text{ in } e_{\text{body}}) \).

First:
\[ \Gamma \vdash \text{exe } e_{\text{val}}, \rho \Rightarrow_{\text{val}} k_{\text{val}} \]

By C.113.

By Lemma C.1.

By C.91 and C.93 and E-Open-Ok.

By C.114 and because \text{inj}(\sigma).

By C.101 and Lemma C.27.

By Lemma C.26.

By C.104 and C.105 on Lemma C.31.

By C.109 and Lemma C.16.

By C.115.

By C.91 and C.93 and E-Open-Ok.
And:
\[ \Gamma \vdash_{exe} \langle \text{e}_{\text{val}}, \rho \rangle \xrightarrow{w} k_{\text{val}} \]
\[ \sigma(w_{\text{val}}) =_\alpha \sigma'(w'_{\text{val}}) : \tau_{\text{val}} \]

Case 1:
\( w = \text{FAULT or } w' = \text{FAULT } \)
\( w = w' = \text{FAULT} \)
\[ \Gamma \vdash_{exe} \langle \text{e}_{\text{all}}, \rho \rangle \xrightarrow{k} \text{FAULT} \]
\[ \Gamma \vdash_{exe} \langle \text{e}_{\text{all}}, \rho \rangle \xrightarrow{k'} \text{FAULT} \]

Case 2:
\( w \neq \text{FAULT and } w' \neq \text{FAULT} \)
Define:
\[ \Gamma_{\text{IH}} \triangleq \Gamma, x : \text{BAtom} \]
Now:
\[ \Gamma_{\text{IH}} \vdash_{exe} \langle \text{e}_{\text{body}}, \rho[x \rightarrow v] \rangle \xrightarrow{w} k_{\text{body}} \]
And:
\[ \Gamma_{\text{IH}} \vdash_{exe} \langle \text{e}_{\text{body}}, \rho'[x \rightarrow v'] \rangle \xrightarrow{w'} k_{\text{body}} \]
But:
\[ \sigma(\rho[x \rightarrow v]) =_\alpha \sigma'(\rho'[x \rightarrow v']) : \Gamma_{\text{IH}} \]
\[ \sigma(w) =_\alpha \sigma'(w') : \tau \]
Furthermore:
\[ \Gamma \vdash_{exe} \langle \text{e}_{\text{all}}, \rho \rangle \xrightarrow{k} w \text{ and } \Gamma \vdash_{exe} \langle \text{e}_{\text{all}}, \rho' \rangle \xrightarrow{k'} w' \]

- Case E-CASE\*:
  WLOG, assume \( \rho(x_{\text{obj}}) = \text{inj}_0(v_0) \) for some \( v_0 \). Then, \( \rho'(x_{\text{obj}}) = \text{inj}_0(v'_0) \) for some \( v'_0 \), by the def. of \( =_\alpha \).

First:
\[ \Gamma, x_0 : \tau_0 \vdash_{exe} \langle e_0, \rho[x_0 \rightarrow v_0] \rangle \xrightarrow{w} k \]
And:
\[ \Gamma, x_0 : \tau_0 \vdash_{exe} \langle e_0, \rho'[x_0 \rightarrow v'_0] \rangle \xrightarrow{w'} k' \]
\( \rho(x_{\text{obj}}) = \text{inj}_0(v_0) \) and \( \rho'(x'_{\text{obj}}) = \text{inj}_0(v'_0) \)
\[ \sigma(v_0) =_\alpha \sigma'(v'_0) : \tau_0 \]
\[ \sigma(\rho[x_0 \rightarrow v_0]) =_\alpha \sigma'(\rho'[x_0 \rightarrow v'_0]) : \Gamma, x_0 : \tau_0 \]
\[ \sigma(w) =_\alpha \sigma'(w') : \tau \]
Furthermore:

\[ \Gamma \vdash_{\text{exe}} \langle \text{case } x \ (x_0 \ e_0) \ (x_1 \ e_1), \rho \rangle \xrightarrow{k} w \]
\[ \Gamma \vdash_{\text{exe}} \langle \text{case } x \ (x_0 \ e_0) \ (x_1 \ e_1), \rho' \rangle \xrightarrow{k'} w' \]

By C.118 and C.119 and E-Case-Left.

- **Case E-If-\(*\):**

  First:

  \[ \rho(x_0) = a \text{ and } \rho'(x_0) = a' \]
  \[ \rho(x_0) = b \text{ and } \rho'(x_0) = b' \]

  \[ \sigma(a) = \alpha \cdot \sigma'(a') : \tau' \text{ and } \sigma(b) = \alpha \cdot \sigma'(b') : \tau' \]

  \[ \sigma(a) = \sigma'(a') \text{ and } \sigma(b) = \sigma'(b') \]

  \[ \sigma(a) = \sigma(b) \text{ iff } \sigma'(a') = \sigma'(b') \]

  \[ a = b \text{ iff } a' = b' \]

  Without loss of generality, assume:

  \[ a = b \]

  Now:

  \[ \Gamma \vdash_{\text{exe}} \langle e_0, \rho \rangle \xrightarrow{k} w \]
  \[ \Gamma \vdash_{\text{exe}} \langle e_0, \rho' \rangle \xrightarrow{k'} w' \]

  \[ w = \alpha \cdot w' : \tau \]

  Because \( \sigma(\rho) = \alpha \cdot \sigma'(\rho') : \Gamma \) on IH.

- **Case E-QLit:** By induction on the structure of \( e \text{lit} \). All cases follow trivially from the definition of \( =_\alpha \), except

  \[ e \text{lit} = \text{prod} (v_i) \text{ and } \tau = \text{Prod}^{\beta_n} (\tau_i \downarrow \beta_i) \]

  which follows from Lemma C.45.

\[ \square \]

**Theorem C.2** (\( \alpha \)-equivalent environments have equivalent termination behavior).

If

\[ \tau = \text{typeof} (\Gamma, e) \]

and \( \rho =_\alpha \rho' : \Gamma \)

and \( \Gamma \vdash_{\text{exe}} \langle e, \rho \rangle \xrightarrow{k} w \)

then

\[ \exists w'. \Gamma \vdash_{\text{exe}} \langle e, \rho' \rangle \xrightarrow{k} w' \text{ and } w =_\alpha w' : \tau \]

**Proof.** By induction on \( k \), and case analysis on \( e \):

- **Case E-Fresh-Ok:** (**fresh** \( x \) in \( e \))

  It suffices to show lines C.133 and C.132.

  First:

  \[ \Gamma, x : \text{BAtom} \vdash_{\text{exe}} \langle e, \rho [x \rightarrow a] \rangle \xrightarrow{k} w \]

  \[ a \notin \text{fa}_\text{env}(\Gamma, \rho) \text{ and } w = \text{FAULT} \lor a \notin \text{fa}(\tau, w) \]

  By inv. of E-Fresh-Ok. (C.129)

  Likewise. (C.130)
\[\rho [x \rightarrow a] =_\alpha \rho' [x \rightarrow a] : \Gamma, x : \text{BAtom} \]

By C.129 and IH. \hspace{1cm} (C.131)

But:

\[\exists u'. \Gamma, x : \text{BAtom} \vdash \text{exec} (e, \rho' [x \rightarrow a]) \Downarrow u'\]

Likewise. \hspace{1cm} (C.132)

\[w =_\alpha u'.:\tau\]

By C.130 on Lemma C.36. \hspace{1cm} (C.133)

\[\Gamma \vdash \text{exec} ((\text{fresh} x \in e), \rho') \xrightarrow{k+1} u'\]

• 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 C.140 and C.138.

First:

\[\Gamma, (x_i : \tau_i)_i \vdash \text{exec} (e, \rho [x_i \rightarrow w_i]) \Downarrow w\]

By inv. of E-Open-Ok. \hspace{1cm} (C.134)

\[\text{fa}_{\text{env}} (\Gamma, \rho) \vdash \text{suff-disj \ prod} (w_i)_i : \tau_{\text{obj}}\]

Likewise. \hspace{1cm} (C.135)

\[w = \text{FAULT} \lor \text{xa(\tau_{\text{obj}}, \prod (w_i)_i)} \not\vdash \text{fa(\tau, w)}\]

Likewise. \hspace{1cm} (C.136)

By C.134 and IH. \hspace{1cm} (C.137)

But:

\[\rho [x_i \rightarrow w_i]_i =_\alpha \rho' [x_i \rightarrow w_i]_i : \Gamma, (x_i : \tau_i)_i\]

Because \(\rho =_\alpha \rho'.:\Gamma\) and by Lemma C.14. \hspace{1cm} (C.138)

\[\exists u'. \Gamma', (x_i : \tau_i)_i \vdash \text{exec} (e, \rho' [x_i \rightarrow w_i]) \Downarrow u'\]

By C.135. \hspace{1cm} (C.139)

\[w =_\alpha u'.:\tau\]

By C.138 on Lemma C.36. \hspace{1cm} (C.140)

By C.136 and C.138. \hspace{1cm} (C.141)

\[\Gamma \vdash \text{exec} ((\text{open} x ((x_i)_i) e), \rho') \xrightarrow{k+1} u'\]

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

First:

\[(x_{\text{frmi}} : \tau_{\text{frmi}})_i \vdash \text{exec} (e, [x_{\text{frmi}} \rightarrow \rho_i (x_{\text{actli}})_i]) \Downarrow w\]

By inv. of E-Call. \hspace{1cm} (C.141)

\[x_{\text{frmi}} \rightarrow \rho (x_{\text{actli}})_i =_\alpha [x_{\text{frmi}} \rightarrow \rho' (x_{\text{actli}})_i] : (x_{\text{frmi}} : \tau_{\text{frmi}})_i\]

Because \(\rho =_\alpha \rho'.:\Gamma\). \hspace{1cm} (C.142)

\[\exists u'. (x_{\text{frmi}} : \tau_{\text{frmi}})_i \vdash \text{exec} (e, [x_{\text{frmi}} \rightarrow \rho' (x_{\text{actli}})_i]) \Downarrow w'\]

By IH on C.141. \hspace{1cm} (C.143)

\[w =_\alpha u'.:\tau\]

Likewise. \hspace{1cm} (C.144)
And:

\[ \Gamma \vdash_{\text{exe}} \left( (\text{if } x i) i \right), \rho^i \xrightarrow{k+1} w' \]

By E-CALL on C.142. (C.144)

- **Case E-If-\*: (if \( x \) equals \( x' e_{\text{yes}} e_{\text{no}} \))** It suffices to show C.145 and C.146.

  First:
  \[
  \begin{align*}
  \rho(x) &= \rho'(x) \quad \text{and} \quad \rho(x') = \rho'(x') \\
  \rho(x) &= \rho'(x') \quad \text{iff} \quad \rho'(x) = \rho'(x') 
  \end{align*}
  \]
  By inv. of Bo-BAtom.
  By simple substitution.

  \[ \Gamma \vdash_{\text{exe}} \left( (\text{if } x \text{ equals } x' e_{\text{yes}} e_{\text{no}}), \rho' \right) \xrightarrow{k+1} w' \]
  By IH and E-If-\*. (C.145)

  And:
  \[ w =_{\alpha} w' : \tau \]
  Likewise. (C.146)

- **Case E-QLit: \( e^{\text{qlit}} \)** We proceed by induction on the structure of \( e^{\text{qlit}} \):

  If \( \Gamma \vdash_{\text{type}} e^{\text{qlit}} : \tau \) and \( \rho =_{\alpha} \rho' : \Gamma \), then \( \llbracket e^{\text{qlit}} \rrbracket_{\rho} =_{\alpha} \llbracket e^{\text{qlit}} \rrbracket_{\rho'} : \tau \)

  All cases except for prod and \( x \) are trivial.

Suppose:

\[
\begin{align*}
  e^{\text{qlit}} &= (\text{prod}^{i_{\alpha}} (e^{\text{qlit}}_{i} \downarrow_{\tau_i} i)) \\
  \forall i. \llbracket e^{\text{qlit}}_{i} \rrbracket_{\rho} &=_{\alpha} \llbracket e^{\text{qlit}}_{i} \rrbracket_{\rho'} : \tau_i \\
  \text{prod} (\llbracket e^{\text{qlit}}_{i} \rrbracket_{\rho} i) &=_{\alpha} \text{prod} (\llbracket e^{\text{qlit}}_{i} \rrbracket_{\rho'} i) : \tau
  \end{align*}
\]

By IH. (C.147)

Or suppose:

\[
\begin{align*}
  e^{\text{qlit}} &= x \\
  \rho(x) &=_{\alpha} \rho'(x) : \tau
  \end{align*}
\]

Because \( \rho =_{\alpha} \rho' : \Gamma \).
Appendix D

Soundness of the deduction system

D.1 Non-occurrence of escape

• E-Fresh

Lemma D.1 (E-Fresh-Fail will not occur).
Let $\Gamma' = \Gamma, x : \text{B Atom}, \tau \vdash \rho' = \rho [x \rightarrow a] [\cdot \rightarrow w]$. If $\Gamma'; \rho' \models F(x) \not\models F(\cdot)$, then $a \notin \text{fa}(\tau, w)$

Proof.
We know:
\[
\|F(x)\|_{\Gamma', \rho'} \not\models \|F(\cdot)\|_{\Gamma', \rho'}
\]
\[
\{a\} \not\models \text{fa}(\tau, w)
\]
\[
a \notin \text{fa}(\tau, w)
\]
By hypothesis.
By def. of $F$ and $\rho'$.
By set arithmetic.

Lemma D.2 ($a$ is fresh for $\Gamma$).
Let $\Gamma' = \Gamma, x : \text{B Atom} and \rho' = \rho [x \rightarrow a]$. If $a \notin \text{fa}(\tau, w)$, and $x \notin \Gamma$, then $\Gamma'; \rho' \models F(x) \not\models F(\cdot)$. 

Proof.
We know:
\[
\|F(x)\|_{\Gamma', \rho'} = \{a\}
\]
\[
\# \text{fa}(\tau, w)
\]
By def. of $F$.
By hypothesis.
\[
\|F(\cdot)\|_{\Gamma', \rho'}
\]
By def. of $F(\cdot)$.

Lemma D.3 (E-Open-Fail will not occur).
Let $\Gamma' = \Gamma, (x_i : \tau_i)_i, \vdash \tau \vdash \rho' = \rho [x_i \rightarrow v_i]_i [\cdot \rightarrow w]$. If $\Gamma'; \rho' \models X_{\text{obj}} (x_i)_i \not\models F(\cdot)$, then $\text{xa}(\tau_{\text{obj}}, \text{prod} (v_i)_i) \not\models \text{fa}(\tau, w)$

Proof. Follows immediately from the definitions of $F$ and $X$. 

86
Lemma D.4 (exposed atoms are fresh for \( \Gamma \)).

Let \( \Gamma' = \Gamma \setminus \{x_i : \tau_i\} \), and \( \rho' = \rho |_{x_i \mapsto v_i} \). If \( \rho(x_{\text{obj}}) = \text{\texttt{prod}}(v_i) \) and \( \forall i \ x_i \notin \Gamma \), then \( \Gamma', \rho' \models \mathcal{X}_{\tau_{\text{obj}}}(x_i) \). \( \Gamma, \rho \models \mathcal{F}(\Gamma) \).

Proof. Follows immediately from the definitions of \( \mathcal{F} \) and \( \mathcal{X} \). \( \Gamma \)

\( \mathcal{E} \)-\texttt{InEq}

Lemma D.5 (Justification of \( \mathcal{I} \)-\texttt{Eq} hypotheses).

Suppose \( a_0 \in \Gamma \) and \( a_1 \in \Gamma \). Then \( \rho(a_0) = \rho(a_1) \) implies \( \Gamma; \rho \models H \Rightarrow \mathcal{F}(a_0) = \mathcal{F}(a_1) \), and \( \rho(a_0) \neq \rho(a_1) \) implies \( \Gamma; \rho \models H \Rightarrow P \wedge \mathcal{F}(a_0) \neq \mathcal{F}(a_1) \).

Proof. Both cases follow immediately from the definition of \( \mathcal{F} \).

D.2 Soundness itself

Terminological note: Except for the phrase “induction hypothesis”, “hypothesis” will always refer to hypothesis predicate of \( \models \) or \( \vdash \) (usually notated with \( H \)). The hypotheses of inference rules will be called “premises”. The consequent predicate of \( \models \) or \( \vdash \) (usually notated with \( P \)) will be called a “consequent”.

Lemma D.6 (Dotlessness of \( H \)).

If \( \Gamma \vdash \text{\texttt{proof}} \{ H \} \vdash \{ P \} \) appears in the derivation of \( \vdash \text{\texttt{proof}} \Gamma \vdash \text{\texttt{ok}} \), then \( \cdot \) does not appear in \( H \).

Proof. By induction over the deduction system. It suffices to observe that first, \( \cdot \) is forbidden from appearing in function preconditions, and second, all deduction system rules that invoke \( \vdash \text{\texttt{proof}} \Gamma \vdash \text{\texttt{ok}} \) only extend \( H \) with terms whose variables are of the form \( x \), which is syntactically unable to be \( \cdot \). \( \Gamma \)

Lemma D.7.

For all sets \( A, B, \) and \( C \), if \( (A \setminus B) \cap C = \emptyset \), then \( A \cap C \subseteq B \).

Proof. Follows from set arithmetic. Intuitively: \( A \) is larger than \( A \setminus B \) by no more than the contents of \( B \). This property is preserved under intersection; therefore \( A \cap C \) differs from \( \emptyset \) by no more than the contents of \( B \). \( \Gamma \)

Lemma D.8 (No names made up).

If \( \Gamma \vdash \text{\texttt{type-env}} \rho \) and \( \Gamma \vdash \text{\texttt{type}} e : \tau \) and \( \Gamma \vdash \text{\texttt{exe}} \langle e, \rho \rangle \Downarrow v \) then \( \text{\texttt{fa}}(\tau, v) \subseteq \text{\texttt{fa}}_{\text{env}}(\Gamma; \rho) \)

Proof. By induction on \( k \).

\( \bullet \) Case (fresh \( x \) in \( e \)):

First:

\( a \notin \text{\texttt{fa}}(\tau, v) \) \hspace{1cm} By \( w_v \neq \text{\texttt{fault}} \) and inv. of \( \text{\texttt{E-Fresh-Ok}} \). \( \Gamma \)

But:

\( \text{\texttt{fa}}(\tau, v) \subseteq \text{\texttt{fa}}_{\text{env}}(\Gamma, x: \text{\texttt{BAtom}}, \rho[x \mapsto a]) \) \hspace{1cm} By \texttt{IH} on \( e \).
\[ = \text{fa}_{\text{env}}(\Gamma, \rho) \cup \{a\} \]
\[ \text{fa}(\tau, v) \subseteq \text{fa}_{\text{env}}(\Gamma, \rho) \]

- **Case (open \(x_{\text{obj}}\ (x_i)_{i} \) \(e_{\text{body}}\)):** Let \(v_{\text{obj}} = \text{prod} (v_i)_{i}\)

  **First:**
  \[ \text{xa}(\tau_{\text{obj}}, v_{\text{obj}}) \neq \text{fa}(\tau, w) \]
  \[ \bigcup_i \text{fa}(\tau_i, v_i) \neq \text{fa}(\tau, w) \]
  \[ \text{fa}(\tau, v) \cap \bigcup_i \text{fa}(\tau_i, v_i) = \emptyset \]
  \[ \text{fa}(\tau, v) \cap \bigcup_i \text{fa}(\tau_i, v_i) \subseteq \text{fa}(\tau_{\text{obj}}, v_{\text{obj}}) \]
  \[ \subseteq \text{fa}_{\text{env}}(\Gamma, \rho) \]

  **But:**
  \[ \text{fa}(\tau, v) \subseteq \text{fa}_{\text{env}}(\Gamma, (x_i : \tau_i), \rho[x_i \mapsto v_i]) \]
  \[ = \text{fa}_{\text{env}}(\Gamma, \rho) \cup \bigcup_i \text{fa}(\tau_i, v_i) \]
  \[ \text{fa}(\tau, v) \subseteq \text{fa}_{\text{env}}(\Gamma, \rho) \cup \text{fa}_{\text{env}}(\Gamma, \rho) \]
  \[ = \text{fa}_{\text{env}}(\Gamma, \rho) \]

- **Case (let \(x\) where \(C\) be \(e_{\text{val}}\) in \(e_{\text{body}}\)):**

  **First:**
  \[ \text{fa}(\tau_{\text{val}}, v_{\text{val}}) \subseteq \text{fa}_{\text{env}}(\Gamma, \rho) \]
  \[ \text{fa}_{\text{env}}(\Gamma, x : \tau_{\text{val}}, \rho[x \mapsto v_{\text{val}}]) = \text{fa}_{\text{env}}(\Gamma, \rho) \]

88
\[ \text{fa}(\tau, v) \subseteq \text{fa}_\text{env}(\Gamma, x : \tau_{\text{val}}, \rho [x \rightarrow v_{\text{val}}]) \]
\[ = \text{fa}_\text{env}(\Gamma, \rho) \]  
By IH on \( e_{\text{body}} \).
By D.3.

- **Case (case \( x (x_0 e_0) (x_1 e_2) \)):** WLOG, consider E-CASE-LEFT.

  First:
  \[ \text{fa}(\tau_0, v_0) = \text{fa}(\Gamma(x), \rho(x)) \]
  \[ \subseteq \text{fa}_\text{env}(\Gamma, \rho) \]  
  By def. of \( \text{fa} \), and by \( \rho(x) = \text{inj}_0(v_0) \).
  By def. of \( \text{fa}_\text{env} \).  
  (D.4)

  But:
  \[ \text{fa}(\tau, v) \subseteq \text{fa}_\text{env}(\Gamma, x : \tau_0, \rho [x \rightarrow v_0]) \]
  \[ = \text{fa}_\text{env}(\Gamma, \rho) \]  
  By IH on \( e_0 \).
  By D.4.

- **Case (if \( x_0 \) equals \( x_1 e_0 e_1 \)):** WLOG, consider E-IF-YES. Follows directly from IH.

- **Case \( e_{\text{eq}} \):** First, by E-QLRT, \( v = \llbracket e_{\text{eq}} \rrbracket_\rho \). We will show that \( \text{fa}(\tau, \llbracket e_{\text{eq}} \rrbracket_\rho) \subseteq \text{fa}_\text{env}(\Gamma, \rho) \) by induction on \( e_{\text{eq}} \).
  - **Case \( e_{\text{eq}} = (\text{prod} \ldots) \) and \( \llbracket e_{\text{eq}} \rrbracket_\rho = \text{prod}(v_i)_i \) and \( \tau = \text{prod}_{\beta_{\text{in}}} (\tau_i \downarrow_{\beta_i})_i \):  
    \[ \text{fa}(\tau, \llbracket e_{\text{eq}} \rrbracket_\rho) = \text{fb}(\tau, \text{prod}(v_i)_i) \cup \text{fr}(\tau, \text{prod}(v_i)_i) \]  
    By def. of \( e_{\text{eq}} \) and \( \text{fa} \).
    \[ = \llbracket \beta_{\text{ex}} \rrbracket \left( \bigcup_i \text{fb}(\tau_i, v_i) \right) \cup \bigcup_i \text{fr}(\tau_i, v_i) \]  
    By def. of \( \text{fb} \) and \( \text{fr} \).
    where \( b = \llbracket \beta_{\text{in}} \rrbracket (\text{fb}(\tau_i, v_i)_i) \).
    \[ \subseteq \llbracket \beta_{\text{ex}} \rrbracket \left( \bigcup_i \text{fb}(\tau_i, v_i)_i \right) \cup \bigcup_i \text{fr}(\tau_i, v_i) \]  
    By set arithmetic.
    \[ \subseteq \bigcup_i \text{fb}(\tau_i, v_i) \cup \bigcup_i \text{fr}(\tau_i, v_i) \]  
    By Lemma C.3.
    \[ = \bigcup_i \text{fa}(\tau_i, v_i) \]  
    By def. of \( \text{fa} \).
    \[ \subseteq \text{fa}_\text{env}(\Gamma, \rho) \]  
    By IH for \( \forall i. v_i \) and def. of \( \text{fa}_\text{env} \).

  - **Case (WLOG) \( e_{\text{eq}} = (\text{inj}_0 \ldots) \) and \( \llbracket e_{\text{eq}} \rrbracket_\rho = \text{inj}_0(v_{\text{sub}}) \) and \( \tau = \tau_{\text{sub}} + \tau_{\text{other}} \):  
    \[ \text{fa}(\tau, \llbracket e_{\text{eq}} \rrbracket_\rho) = \text{fa}(\tau, v_{\text{sub}}) \]  
    By def. of \( \text{fa} \), \( \text{fb} \), and \( \text{fr} \).
    \[ \subseteq \text{fa}_\text{env}(\Gamma, \rho) \]  
    By IH for \( v_{\text{sub}} \).

  - **Case \( e_{\text{eq}} = x \):** By T-VAR, \( x \in \Gamma \). By definition of \( \text{fa}_\text{env} \), \( \text{fa}(\tau, \llbracket e_{\text{eq}} \rrbracket_\rho) \subseteq \text{fa}_\text{env}(\Gamma, \rho) \).

\[ \square \]

**Lemma D.9** (No names made up (\( \vdash \) version)).

If \( \Gamma \vdash \text{type-env} \rho \) and \( \Gamma \vdash _{\text{exe}} (e, \rho) \models v \), then \( \Gamma, \vdash : \tau; \rho [\cdot \rightarrow v] \models \mathcal{F}(\cdot) \subseteq \mathcal{F}_e(\Gamma) \)

**Proof.** Follows from Lemma D.8.

\[ \square \]

89
The following utility lemmas will help us maintain the induction hypothesis of the soundness theorem:

**Lemma D.10** (H′ eliminability).
If \( \Gamma; \rho \vdash H \Rightarrow H' \) and \( \Gamma; \vdash \tau; \rho[\cdot \rightarrow w] \vdash H \land H' \Rightarrow P \) then \( \Gamma; \vdash \tau; \rho[\cdot \rightarrow w] \vdash H \Rightarrow P \).

**Proof.** Assume (for the sake of the conclusion we desire) that \( \Gamma; \vdash \tau; \rho[\cdot \rightarrow w] \vdash H. \) By Lemmas D.12 and D.6, we know \( \Gamma; \vdash \tau; \rho[\cdot \rightarrow w] \vdash H \land H', \) and therefore \( \Gamma; \vdash \tau; \rho[\cdot \rightarrow w] \vdash P. \) By our assumption, \( \Gamma; \vdash \tau; \rho[\cdot \rightarrow w] \vdash H \Rightarrow P. \) □

**Lemma D.11** (P eliminability).
If \( \Gamma; \vdash \tau; \rho[\cdot \rightarrow w] \vdash H \Rightarrow P \land P' \) then \( \Gamma; \vdash \tau; \rho[\cdot \rightarrow w] \vdash H \Rightarrow P. \)

**Proof.** Follows from \( \land \)-elimination in the metalogic. □

The following lemmas are standard strengthening and weakening results:

**Lemma D.12** (Strengthening of \( \Gamma \) and \( \rho \) in \( \models_{hyp} \)).
If \( \Gamma; \rho \vdash H \Rightarrow P \) and \( z \) is free for \( H \) and \( P \), then \( \forall v, \tau, \Gamma; \vdash \tau; \rho[z \rightarrow v] \vdash H \Rightarrow P \).

**Proof.** Assume that \( \Gamma; z: \tau; \rho[z \rightarrow v] \vdash H \) (for arbitrary \( v \) and \( \tau \)). An inductive argument on the structure of \( H \) shows that \( \Gamma; \rho \vdash H. \) Then, \( \Gamma; \rho \vdash P. \)

We can conclude that \( \forall v, \tau, \Gamma; \vdash \tau; \rho[z \rightarrow v] \vdash H \Rightarrow P. \) □

**Lemma D.13** (Strengthening of \( \Gamma \) in \( \models_{hyp} \)).
If \( \Gamma \vdash H \Rightarrow P \) and \( z \) is free for \( H \) and \( P \), then \( \forall \tau, \Gamma; z: \tau \vdash H \Rightarrow P \).

**Proof.** Assume that \( \forall \rho, \Gamma; z: \tau; \rho \vdash H \) (for arbitrary \( \tau \)). An inductive argument on the structure of \( H \) shows that \( \Gamma; \rho \vdash H. \) Then, \( \Gamma; \rho \vdash P. \)

We can conclude that \( \forall \tau, \Gamma; z: \tau \vdash H \Rightarrow P. \) □

**Lemma D.14** (Weakening of \( \Gamma \) and \( \rho \) in \( \models_{hyp} \)).
If \( \Gamma; z: \tau; \rho[z \rightarrow v] \vdash H \Rightarrow P \) and \( z \) is free for \( H \) and \( P \), then \( \Gamma; \rho \vdash H \Rightarrow P \).

**Proof.** Assume that \( \Gamma; \rho \vdash H. \) An inductive argument on the structure of \( H \) shows that \( \Gamma; z: \tau; \rho[z \rightarrow v] \vdash H. \) Then, \( \Gamma; z: \tau; \rho[z \rightarrow v] \vdash P. \)

We can conclude that \( \Gamma; \rho \vdash H \Rightarrow P. \) □

**Theorem D.1** (Soundness of the deduction system).

\[
\text{If } \begin{array}{l}
\Gamma \vdash_{type} (fD_i)_{i} & e_{\text{main}} : \tau_{\text{main}} \\
\Gamma \vdash_{proof} (fD_i)_{i} & e_{\text{main}} \text{ ok} \\
\Gamma \vdash_{exe} (fD_i)_{i} & e_{\text{main}} \xrightarrow{k} w_{\text{final}} \\
\text{then } w_{\text{final}} \neq \text{FAULT}
\end{array}
\]

**Proof.** Next, we proceed by induction on \( k \) in the following statement

\[
\begin{array}{l}
\text{If } \Gamma \vdash_{exe} (e, \rho) \xrightarrow{w} \\
\text{and } \exists H, P, \Gamma \vdash_{proof} (H) e \in \{P\} \\
\text{and } \exists \tau, \Gamma \vdash_{type} e : \tau \\
\text{then } w \neq \text{FAULT} \text{ and } \Gamma; \vdash \tau; \rho[\cdot \rightarrow w] \vdash H \Rightarrow P
\end{array}
\]

We will ensure the well-foundedness of this argument by only relying on the induction hypothesis for \( k' < k \) (which is to say, for prior execution steps).
Because of the structure of the induction hypothesis, each case will consist of two parts (though in many cases one of them is trivial). Part (1) will consist of using the deduction system to justify that execution doesn’t fault.

Part (2) of each case is the justification of \( \Gamma, \vdash \tau; \rho' \to w \models H \Rightarrow \neg P \). Lemmas D.10 and D.11 will be used extensively for this.

- **Case E-Fresh-\( \ast \):** \( e = (\text{fresh } x \text{ in } c_{\text{body}}) \). Let \( \Gamma' = \Gamma, x : \text{BAtom} \) and \( \rho' = \rho [x \to a] \). Let \( P_e = P \land F(x) \land F(\cdot) \)

  We have:

  \( a \notin \\text{fa}_\text{env}(\Gamma, \rho) \)

  We have:

  \( w \neq \text{FAULT} \)

  \( \Gamma', \vdash \tau; \rho' [\to w] \models H \land F(x) \land F(\cdot)(\Gamma) \Rightarrow P_e \)

   (Part 1)

  \( a \notin \\text{fa}(\tau, w) \)

  \( \Gamma \vdash_{\text{exe}} \langle e, \rho \rangle \models w \neq \text{FAULT} \)

   (Part 2)

- **Case E-Open-\( \ast \):** \( e = (\text{open } x_{\text{obj}} ((x_i)_i) \text{ in } c_{\text{body}}) \). Let \( \Gamma' = \Gamma, (x_i : \tau_i)_i \) and \( \rho' = \rho [x_i \to v_i]_i \). Let \( P_e = P \land X_{\text{var}} (x_i)_i \land F(\cdot) \)

  We have:

  \[ \left( \bigcup_i \text{fb}(\tau_i, v_i) \backslash \text{fb}(\tau_{\text{obj}}, \text{prod } (v_i)_i) \right) \land \text{fa}_\text{env}(\Gamma, \rho) \]

  \( \text{xa}(\tau_{\text{obj}}, \text{prod } (v_i)_i) \land \text{fa}_\text{env}(\Gamma, \rho) \)

  We have:

  \( w \neq \text{FAULT} \)

  \( \Gamma', \vdash \tau; \rho' [\to w] \models H \land X_{\text{var}} (x_i)_i \land F(\cdot)(\Gamma) \Rightarrow P_e \)

   (Part 1)

  \( \text{xa}(\tau_{\text{obj}}, \text{prod } (v_i)_i) \land \text{fa}(\tau, w) \)

  \( \Gamma \vdash_{\text{exe}} \langle e, \rho \rangle \models w \neq \text{FAULT} \)

   (Part 2)

- **Case E-Open-\( \ast \)**: \( e = (\text{open } x_{\text{obj}} ((x_i)_i) \text{ in } c_{\text{body}}) \). Let \( \Gamma' = \Gamma, (x_i : \tau_i)_i \) and \( \rho' = \rho [x_i \to v_i]_i \). Let \( P_e = P \land X_{\text{var}} (x_i)_i \land F(\cdot) \)

  We have:

  \[ \left( \bigcup_i \text{fb}(\tau_i, v_i) \backslash \text{fb}(\tau_{\text{obj}}, \text{prod } (v_i)_i) \right) \land \text{fa}_\text{env}(\Gamma, \rho) \]

  \( \text{xa}(\tau_{\text{obj}}, \text{prod } (v_i)_i) \land \text{fa}_\text{env}(\Gamma, \rho) \)

  We have:

  \( w \neq \text{FAULT} \)

  \( \Gamma', \vdash \tau; \rho' [\to w] \models H \land X_{\text{var}} (x_i)_i \land F(\cdot)(\Gamma) \Rightarrow P_e \)

   (Part 1)

  \( \text{xa}(\tau_{\text{obj}}, \text{prod } (v_i)_i) \land \text{fa}(\tau, w) \)

  \( \Gamma \vdash_{\text{exe}} \langle e, \rho \rangle \models w \neq \text{FAULT} \)

   (Part 2)

  \( \Gamma'; \rho' \models X_{\text{var}} (x_i)_i \land F(\cdot)(\Gamma) \)

  \( \Gamma', \vdash \tau; \rho' [\to w] \models H \Rightarrow P_e \)

  \( \Gamma', \vdash \tau; \rho' [\to w] \models H \Rightarrow P_e \)

  \( \Gamma, \vdash \tau; \rho [\to w] \models H \Rightarrow P \)
• **Case E-CALL**: \( e = (f \ (x_{\text{actual},i})) \) and \( k' = k + 1 \).

Let \( e_{\text{body}} = \text{body}(f) \) and \( \Gamma_f = (x_{\text{formal},i} : \tau_{\text{formal},i}) \), and \( \rho_f = [x_{\text{formal},i} \rightarrow \rho_i(x_{\text{actual},i})]_i \).

Let \( \Gamma_i = (x_{\text{actual},i} : \tau_{\text{formal},i}) \), and \( \rho_i = [x_{\text{actual},i} \rightarrow \rho_i(x_{\text{actual},i})] \), and \( C_{\text{pre},i} = \text{pre}(f) [x_{\text{actual},i}, x_{\text{formal},i}]_i \), and \( \text{Cpost}_i = \text{post}(f) [x_{\text{actual},i}, x_{\text{formal},i}]_i \).

We have:

\[
\Gamma_f \vdash \text{exe} (e_{\text{body}}, \rho_f) \Downarrow \Downarrow w
\]

We have:

\[
\Gamma_f \vdash \text{type} e_{\text{body}} : \tau_0
\]

We have:

\[
\Gamma_f \vdash \text{proof } \{ \text{pre}(f) \} e_{\text{body}} \{ \text{post}(f) \}
\]

We have:

\[
\Gamma \models H \Rightarrow C_{\text{pre},i}
\]

We have:

\[
\Gamma_i : \tau \models H \land \mathcal{F}(\cdot) \subseteq E_e(\Gamma_i) \land C_{\text{post},i} \Rightarrow P
\]

Now:

\[
w \neq \text{FAULT}
\]

\[
\Gamma_f, : \tau ; \rho_f \models [\rightarrow w] \Rightarrow \text{pre}(f) \Rightarrow \text{post}(f)
\]

(Part 1)

\[
\Gamma \vdash \text{exe} (e, \rho) \Downarrow \Downarrow \Downarrow w \neq \text{FAULT}
\]

(Part 2)

\[
\Gamma_i, : \tau ; \rho_i \models [\rightarrow w] \Rightarrow C_{\text{pre},i} \Rightarrow C_{\text{post},i}
\]

But:

\[
\text{dom}(\Gamma_i) \subseteq \text{dom}(\Gamma) \text{ and dom}(\rho_i) \subseteq \text{dom}(\rho)
\]

So:

\[
\Gamma_i, : \tau ; \rho \models [\rightarrow w] \Rightarrow C_{\text{pre},i} \Rightarrow C_{\text{post},i}
\]

But:

\[
\Gamma_i, : \tau \models H \Rightarrow C_{\text{pre},i}
\]

So:

\[
\Gamma_i, : \tau ; \rho \models [\rightarrow w] \Rightarrow H \Rightarrow C_{\text{post},i}
\]

\[
\Gamma, : \tau ; \rho \models [\rightarrow w] \Rightarrow \mathcal{F}(\cdot) \subseteq E_e(\Gamma) \land C_{\text{post},i}
\]

\[
\Gamma, : \tau ; \rho \models [\rightarrow w] \Rightarrow P
\]

• **Case E-LET**: \( e = (\text{let } x \text{ where } C \text{ be } e_{\text{val}} \text{ in } e_{\text{body}}) \).

Let \( \Gamma' = \Gamma, x : \tau_{\text{val}}, : \tau \text{ and } \rho' = \rho [x \rightarrow v_{\text{val}}] [\rightarrow w] \).

We have:

E-LET-FAIL does not occur

\[
\text{IH on } e_{\text{val}} \text{ premises of E-LET-*}, \text{T-LET and P-LET.}
\]

\[92\]
\[\Gamma, \cdot \vdash_{\text{val}} \rho \vdash_{\text{val}} H \Rightarrow C\]

We have:

Let \(w \not\in \text{FAULT}\)

\[\Gamma', \rho' \vdash C \frac{x/} \land F(x) \subseteq F_{e}(\Gamma) \Rightarrow P\] (Part 1)

\[\Gamma \vdash_{\text{ex}} \langle e, \rho \rangle \overset{k_{\text{val}} + k_{\text{body}} + 1}{\longrightarrow} w \not\in \text{FAULT}\]

\[\Gamma, \cdot \vdash_{\text{val}} \rho \vdash_{\text{val}} F(\cdot) \subseteq F_{e}(\Gamma)\]

\[\Gamma, \cdot \vdash_{\text{val}} \rho \vdash_{\text{val}} H \Rightarrow C \land F(\cdot) \subseteq F_{e}(\Gamma)\]

\[\Gamma, x: \tau_{\text{val}}; \rho \vdash x \vdash_{\text{val}} F(x) \subseteq F_{e}(\Gamma)\]

\[\Gamma', \rho' \vdash H \Rightarrow C \frac{x/} \land F(x) \subseteq F_{e}(\Gamma)\]

\[\Gamma', \rho' \vdash H \Rightarrow P\]

\[\Gamma, \cdot \vdash_{\text{val}} \rho \vdash_{\text{val}} H \Rightarrow P\]

\[\cdot\text{Case E-Case-}\star \cdot e = (\text{case } x \cdot (x_0 \cdot e_0) \cdot (x_1 \cdot e_1)) \cdot \text{WLOG, consider E-Case-LEFT.}\]

Let \(\Gamma' = \Gamma, x: \tau_0\) and \(\rho' = \rho \vdash_{\text{val}} w\)

We know:

Let \(w \not\in \text{FAULT}\)

\[\Gamma', \cdot \vdash_{\text{val}} \rho \vdash_{\text{val}} H \Rightarrow P\] (Part 1)

\[\Gamma \vdash_{\text{ex}} \langle e, \rho \rangle \overset{k_{\text{val}} + k_{\text{body}} + 1}{\longrightarrow} w \not\in \text{FAULT}\]

\[\Gamma, \cdot \vdash_{\text{val}} \rho \vdash_{\text{val}} H \Rightarrow P\] (Part 2)

\[\Gamma, \cdot \vdash_{\text{val}} \rho \vdash_{\text{val}} H \Rightarrow P\]

\[\cdot\text{Case E-\text{IrEq}: } e = (\text{if } x_0 \text{ equals } x_1 \cdot e_0) \text{. Let } H_0 = H \land F(x_0) = F(x_1) \text{ and } H_1 = H \land F(x_0) \not\subseteq F(x_1).\]

We know:

Let \(w \not\in \text{FAULT}\)

if \(\rho(x_0) = \rho(x_1), \text{ then } \Gamma, \cdot \vdash_{\text{val}} \rho \vdash_{\text{val}} H_0 \Rightarrow P\)

if \(\rho(x_0) \not= \rho(x_1), \text{ then } \Gamma, \cdot \vdash_{\text{val}} \rho \vdash_{\text{val}} H_1 \Rightarrow P\)

\[\Gamma \vdash_{\text{ex}} \langle e, \rho \rangle \overset{k}{\longrightarrow} w \not\in \text{FAULT}\] (Part 1)

\[\Gamma, \cdot \vdash_{\text{val}} \rho \vdash_{\text{val}} H \Rightarrow P\]

\[\cdot\text{IH on inv. of E-IrEq-\text{\textdagger} \text{, T-IrEq, P-IrEq.}}\]

Likewise (E-IrEq-Yes). (D.27)

Likewise (E-IrEq-No). (D.28)

By D.26.

Lemma D.5 applied to D.27 and D.28.
• Case $E\text{-QLtr}: e = e_{\text{qlit}}$.

(Part 1)
$\Gamma \vdash_{\text{exe}} \langle e_{\text{qlit}}, \rho \rangle \overset{k}{\Rightarrow} v$

(Part 2)
$\Gamma, ::\tau \vdash H \land (\cdot \equiv e_{\text{qlit}}) \Rightarrow P$
Also:
$v = \llbracket e_{\text{qlit}} \rrbracket,_{\rho}$
$\Gamma, ::\tau; \rho [\cdot \rightarrow v] \vdash \cdot \equiv e_{\text{qlit}}$
$\Gamma, ::\tau; \rho [\cdot \rightarrow v] \vdash H \Rightarrow P$

By definition of $E\text{-QLtr}$.

By inv. of P-$\text{QLtr}$.  (D.29)

By inv. of $E\text{-QLtr}$.

By definition of $\equiv$.

By D.29 and Lemma D.10.
Bibliography


