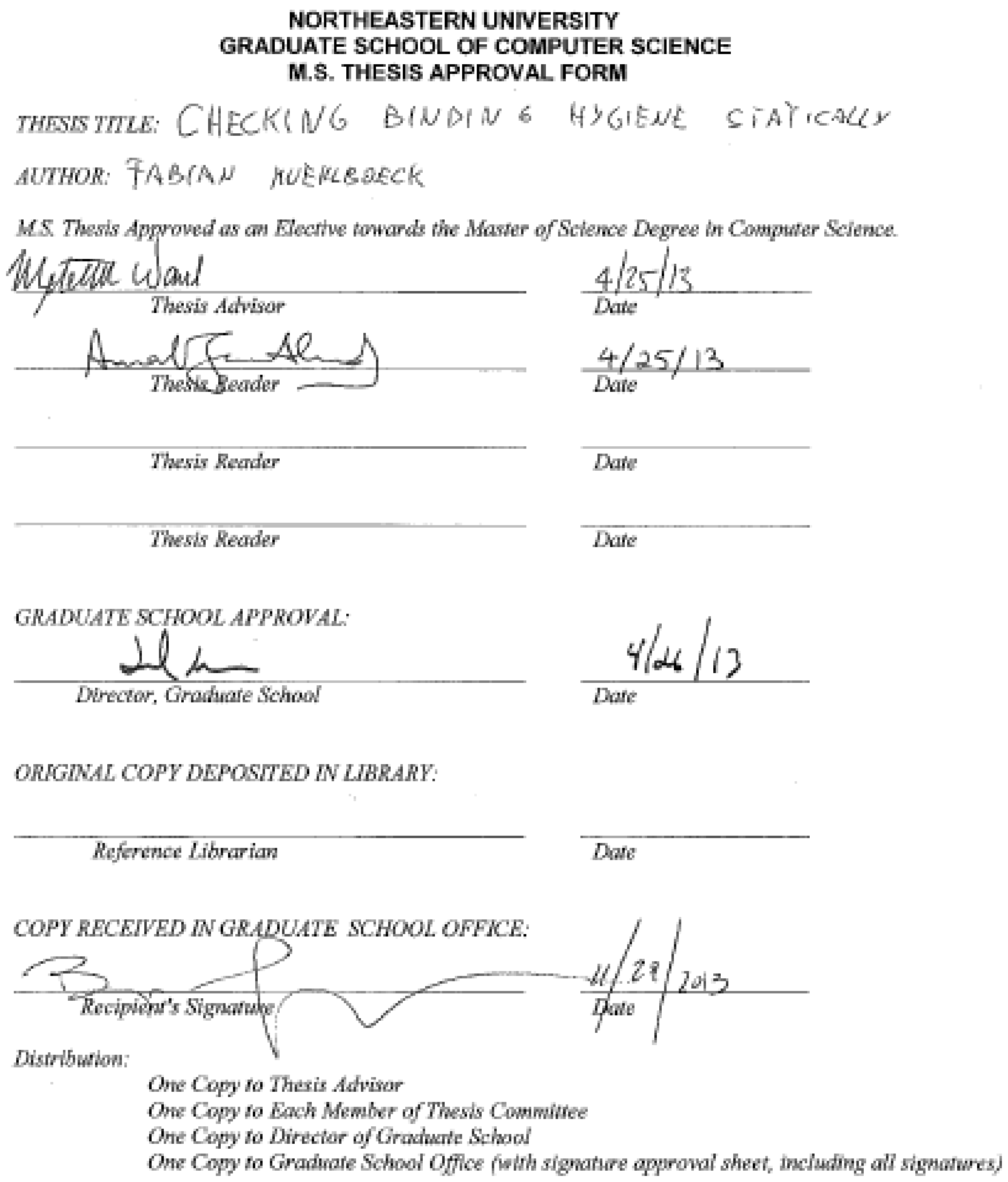
Checking Binding Hygiene Statically

With few annotations and with meaningful explanations for errors

Fabian Muehlboeck
April 2013



NORTHEASTERN UNIVERSITY
GRADUATE SCHOOL OF COMPUTER SCIENCE
M.S. THESIS APPROVAL FORM

THESIS TITLE: CHECKING BINDING HYGIENE STATICALLY

AUTHOR: FABIAN MUEHLBOECK

M.S. Thesis Approved as an Elective towards the Master of Science Degree in Computer Science.

[illegible]	4/25/13
Thesis Advisor	*Date*
[illegible]	4/25/13
Thesis Reader	*Date*
Thesis Reader	*Date*
Thesis Reader	*Date*

GRADUATE SCHOOL APPROVAL:

[illegible]	4/26/13
Director, Graduate School	*Date*

ORIGINAL COPY DEPOSITED IN LIBRARY:

Reference Librarian	*Date*

COPY RECEIVED IN GRADUATE SCHOOL OFFICE:

[illegible]	11/29/2013
Recipient's Signature	*Date*

Distribution:
One Copy to Thesis Advisor
One Copy to Each Member of Thesis Committee
One Copy to Director of Graduate School
One Copy to Graduate School Office (with signature approval sheet, including all signatures)

M.S. Thesis in Computer Science

College of Computer and Information Science

Northeastern University, Boston, MA

Thesis Advisor: Mitchell Wand

Thesis Reader: Amal Ahmed

Checking Binding Hygiene Statically

Abstract

Macros are a powerful facility for various applications, from simple syntax extensions to creating domain specific languages. However, they have suffered from the fact that the structure of variable bindings in a program is only implicitly encoded in the program text—because of this, simple program transformations may lead to results not intended by either the macro writer or user by violating hygiene. We present the core details of a developing system (by Stansifer and Wand) to statically ensure the preservation of α-equivalence and therefore hygiene.

Our contribution comes in two parts. The first is that we finished the implementation of the system by creating an interface to an SMT-solver to discharge proof obligations generated by the system.

The second part concerns the usability of this system: we implemented a language with ML-like syntax similar to that found in the implementations of related work, and are able to show that the burden on the programmer to write down annotations is comparable to similar systems because many annotations can be inferred. We also show how errors can be explained to the user when proof obligations cannot be discharged.

Knowledge prerequisites

We assume that the reader is familiar with the basic ideas of the λ-calculus and any functional programming language. Some sections require a basic knowledge of type theory for full understanding. Both of these prerequisites are explained in the first few chapters of Pierce's Textbook on Types and Programming Languages [1].

Checking Binding Hygiene Statically

Acknowledgements

I would like to express my sincerest thanks to all the people who have supported me in in writing this thesis, and more broadly, in my master studies here at Northeastern. Thanks to Mitchell Wand, who invited me to join the project that this thesis is based on, and then listened to my ideas and pointed out the right direction. His guidance was invaluable to me. Thanks to Amal Ahmed, whose detailed comments on drafts on this thesis helped me to improve the text by a huge margin. Thanks to Paul Stansifer, on whose work my whole project was based on and who spent numerous hours explaining the technical background to me and helping me debug my programs. Thanks to Ian Johnson, who provided lots of additional background information and ideas.

Thanks to all the people who made studying in Boston such a great experience. My student colleagues, who are both nice to work with and talk to, and are always willing to help when help is needed. Thanks to the great people at the Northeastern Programming Research Lab, who were not only fun to hang around with, but also offered lots of helpful advice and comments.

Finally, thanks to the people who enabled me to come to Northeastern. Thanks to the Fulbright Program—and therefore the citizens of Austria and the United States—and especially the Austrian Fulbright Commission, who gave me this great opportunity. Thanks to Franz Puntigam, who made me see how interesting the field of programming languages is. Special thanks to my parents and the rest of my family, who have loved and supported me for more than a quarter of a century now.

And special thanks to you, Sabine, for sticking with me even though I am away from you most of the time.

CONTENT

1 Macros, Binding and Hygiene

1.1 Macros

Macro systems provide the possibility to extend a programming language with new syntactic constructs. While there are many different macro systems with different capabilities, at their core they all look for patterns in the source code of a program and replace it according to some rules. This process is called *macro expansion*, and should typically end with a program in the unextended base language, although there may be several passes of macro expansion until we get there.

Macro systems are especially prominent in the family of LISP-like programming languages, but other languages offer macro systems too (e.g. LaTeX or C) or have extensions that offer macros (e.g. MacroML [2] for OCaml, or Maya [3] for Java). Whatever language is using a macro system, if it uses names for variables, then macros may unintentionally introduce name clashes. In order to understand what that means, we will briefly explain what binding and scope means, and what difficulties we face when we want to write syntax-transforming functions. This will lead to the definition of hygiene, a property of macros that rules out name clashes.

1.2 Binding and Scope

Almost every programming language features some notion of *binding*, which means assigning names to objects[1] and referencing those objects by their name. These languages come with *scoping* rules that determine which names are accessible and which names reference which objects. We say that a name is *in scope* in an expression if there is a *declaration* somewhere that *binds* references of that name in that expression. Conversely, the *scope* of a declaration is the part of a program where references to its name are bound to it. The scoping rules have to specify what happens if the scopes of two declarations with the same name would overlap. This can be seen as an error, or one of the declarations must get precedence and therefore *shadow* the other. If shadowing leads to references being bound by a different declaration than the one a programmer intended, we call this a *name clash*, or *unintended capture*.

In general, scoping rules come in two flavors: static and dynamic. In the latter, bindings are dependent on the control flow of the program at run-time, whereas in the former, bindings depend on the structure of the program text.

Static scoping enables *local reasoning* about bindings, which is generally seen as a desirable property, and most languages follow static scoping rules. To reason locally about bindings, programmers need to know two things about their language of choice: the scoping rules, and which parts of the language introduce bindings (that is: what parts of the code are declarations and what parts of those declarations are names). Therefore, following static scoping

[1] Here, object is used as a term for language-level objects, such as types and variables.

```
(define-syntax query/default
  (syntax-rules ()
    [(query/default q d)
     (let ([s q])
       (if s (process-query s) d))]))
```

Example 1.1 – The query/default-macro

rules as a programmer is usually very easy. If name clashes arise, they can be identified quickly by renaming one of the conflicting declarations and all the references that are supposed to refer to it.

1.3 Naïve Macro Expansion and Unintended Captures

Naïve macro expansion is the simple process of searching for patterns in a program and converting matching expressions according to some rules. As binding is only implicitly encoded in the syntactic representation of a program, the macro expansion is unaware of it. In principle, all the macro expansion does is reorder some symbols. In the following, we will explain how unintended captures can occur when using naïve macro expansion. The problem here is that programmers do not see the code resulting from the expansion (and also do not want to). They rather still want to reason locally about their source code. This is a motivating example for many works on macro systems [4] [5] [6]. Herman gives a very detailed explanation [7]. This is not the only class of problems macro systems face. In some cases, programmers would like to ensure that none of the introduced references are free in the result [8].

1.3.1 Outside references

Example 1.1 shows a macro definition in Racket syntax. Assume there is a function `process-query` that takes something we will call a query and returns some result. The `query/default-`

macro takes two arguments, a query expression and a default expression. We assume that the query expression returns either a legal query or `false`. It may also have a side-effect. The function `process-query` does not accept `false` as argument, so `query/default` guards against the `false` case and evaluates the default expression if the query expression evaluates to `false`. Binding the result of the query expression to an intermediate variable in the `let`-expression ensures that the query expression is evaluated (and hence its side-effect is caused) only once. Example 1.2 shows how this macro can be naïvely expanded, while Example 1.3 shows how easily naïve macro expansion can cause trouble:

```
(query/default x y) => (let ([s x]) (if s (process-query s) y))
```

Example 1.2 – Naïve expansion of the query/default-macro

```
(query/default x s) => (let ([s x]) (if s (process-query s) s))
```

Example 1.3 – A problematic case of naïve macro expansion

In Example 1.3, the second argument to the macro is just the identifier `s`, which presumably is a bound variable that will contain the actual default value when the program is run. However, regardless of what value the first argument will evaluate to, the default value will never influence the result of the expression the macro expands to. The variable `s` no longer references the binding it was supposed to refer to, because it got captured by the new binding introduced by the macro.

1.3.2 Outside bindings

The user of a macro is not the only one who may have his bindings destroyed by unintended captures. The macro itself is not safe, either. Example 1.4 shows what happens if the user of a macro defines a variable with a name that is referenced in the macro. Suddenly, the reference to `process-query` inside the macro body is not a reference to the function that `query/default` is supposed to guard, but to whatever the user bound it to.

```
(let ([process-query (get-query “x.exe”)])
  (query/default process-query “Don’t know”))
=>
(let ([process-query (get-query “x.exe”)])
  (let ([s process-query]) (if s (process-query s) “Don’t know”)))
```

Example 1.4 – The binding of a reference in the macro can change, too

1.3.3 Syntactic Keywords

Some languages that treat identifiers of both core constructs and user-defined variables identically may suffer from those two problems in an additional way: those core constructs could be captured, too. For example, in Racket, nothing prevents the user of the `query/default` macro from re-binding the name `if` and thus capturing the `if` in the macro. Similarly, a macro could also re-bind `if`, and if an argument to the macro contains an `if`, it might get captured by the definition of the macro.

1.4 Hygiene

Hygiene is a term coined by Kohlbecker et al. [4]. They solve the problem of unintended captures of outside references (see section 1.3.1). To that end, they present a macro expansion algorithm that satisfies what they call the *Hygiene Condition for Macro Expansion:*

> *Generated identifiers that become binding instances in the completely expanded program must only bind variables that are generated at the same transcription step.*

They also already state a connection between hygiene and α-equivalence, by continuing with the note:

From the λ-calculus, one knows that if the hygiene condition does not hold, it can be established by an appropriate number of α-conversions.

By transcription step, they mean the expansion of one particular macro. Since the expansion of a macro can yield another macro invocation, expansion must continue until no more macro invocations are present. As we saw in the previous sections, macros may introduce new variables and bindings. If those may only bind variables generated in the same step, they cannot bind variables that already were in the code, thus preventing unintentional capture as described above. This is achieved with a simple idea: number the transition steps consecutively, and assign every variable the number of the transition step where it first appears. That means that even if a macro introduces a variable with a name that also appears in the code that uses the macro, the names have different timestamps. After everything is expanded, the binding structure can be explored and variables with the same name and timestamp can be α-renamed together.

The major drawbacks of the algorithm above are that its time complexity is quadratic in the size of the input program, and that it does not solve the second problem we presented: the unintended capture of outside bindings (see section 1.3.2). Clinger and Rees [5] combine the above article with a concept called syntactic closures [9] by carefully constructing syntactic environments to distinguish user variables from macro variables. They improve the running time (now linear in the size of the input program) and solve the problem of outside bindings. However, the set of macros that one may write with this system (syntax-rules) is limited. There is a low-level mechanism to write potentially unhygienic macros, but it requires programmers to make all renaming explicit, making it cumbersome to use.

Dybvig et al. [6] devised an algorithm that also builds on the one by Kohlbecker et al. [4], but also exhibits better running time and handles more errors, and even includes a source-tracking system that relates generated code to the original code, which is useful for debugging. For a long time, hygiene in Scheme macros was defined by what this algorithm did.

Only recently Herman and Wand explored a formal definition of hygiene based on α-equivalence [10].

1.5 α-Equivalence and Related Work

α-equivalence is one of the core concepts of the field of programming languages. Two programs are said to be α-equivalent if they only differ in the names of their bound variables. α-equivalence is an important concept for hygiene, because the whole process of renaming variables to avoid clashes is based on it.

There are various techniques to simplify reasoning about α-equivalence, notably De Bruijn indices [11] and FM-sets [12]. The idea of the former is to not use names for variables, but numbers that represent the depth of a reference relative to its binding occurrence. That way α-equivalence is the same as syntactic equivalence. While this concept is very machine-friendly, almost every paper that mentions De Bruijn indices stresses their lack of readability (there are exceptions—McBride and McKinna, for example argue that a mixed representation of De Bruijn indices and names can be useful for reasoning about syntax manipulation [13]).

Gabbay and Pitts [12] established the link between FM-sets and α-equivalence. Intuitively, an FM-set represents all infinitely many α-equivalent versions of some syntax tree and has nice mathematical properties that enable reasoning about binding, freshness, and techniques such as structural induction and recursion. Together with Shinwell, they then devised a language called FreshML [14], which used the theory of FM-sets to reason about the freshness of names and detect when variables introduced by a syntax transformation might escape and violate the intended binding structure of the program. To that end, FreshML provided a type system for terms where the binding structure was encoded. While its original version provided static checks for FreshML programs (that is, they obtained guarantees before actually running the syntax transformation), they found that this static system was too restrictive in that it rejected many valid programs. Later versions of FreshML would only use the type system to avoid naming conflicts dynamically (that is, when an actual program would be transformed).

Over the years, many different systems with types that contain binding specifications have been presented, with different goals, but all similar in the key role that α-equivalence plays. Ott [15] is a tool to specify languages in a style that looks rather informal, and compiles the specification to LaTeX and theorem prover/proof assistant definitions. Ott features binding specifications and generates boilerplate substitution definitions that respect α-equivalence. Similarly, Weirich et al. [16] worked on binding specifications to avoid boilerplate code when implementing compilers and interpreters and to support constructs of dependently typed languages.

FreshML also spawned more work such as Fresh Objective Caml [17] and pure FreshML by Pottier [18]. The latter revisited static checking[1] of a syntax-transforming program to ensure that variables that are introduced by it are eventually bound, and, more importantly, that two α-equivalent programs are mapped to α-equivalent output—which is the definition of hygiene following Herman [7]—although it has not been formally proven for pure FreshML.

Recently, Stansifer and Wand [19] have been working on a system that is similar to pure FreshML, but features richer binding specifications while offering a simpler semantics, and they are currently working on the formal proof of the preservation of α-equivalence. We will discuss this system in more detail in the following chapters.

1.6 Organization

The rest of this work is organized as follows: in chapter 2, we present Romeo [19], Stansifer and Wand's language featuring binding specifications and a proof system designed to prove that macros written in it preserve α-equivalence and are therefore hygienic. In chapter 3, we discuss several observations on Romeo and extensions that follow from these observations. In chapter 4, we discuss Romeo-L, our implementation of a more concise and useable syntax

[1] In fact, Shinwell, Pitts and Gabbay [14] give an example (Normalization by Evaluation) which is rejected by FreshMLs static version (called FreshML 2000), but accepted by its dynamic version. However, it can be statically checked in pure FreshML and is accepted. Appendix I contains our version of this example.

as well as the implementation of the link between the proof system and the SMT solver that is needed to discharge proof obligations generated by the proof system. We conducted some tests and experiments with this new system and present the results in chapter 5, after which we will conclude and discuss future work.

2 Romeo

In this chapter we will present the features of Romeo [19], a language for macros that features binding specifications and a type/proof system designed in such a way that admitted macros preserve α-equivalence. It is important to note that Romeo is not the work of the author of this thesis and that it is work in progress. This may be the first presentation of it, based on its current status. Also, the presentation is not complete—we will only present the syntax and the proof rules in detail, while giving an intuition on the type system and execution rules. Romeo is very similar to Pottier's pure FreshML [18], and we will discuss the similarities and differences at the end of this chapter.

2.1 Definitions

In this and the following chapters, we will talk about *meta-languages* for programs that transform programs in *object languages*. The meta-languages use the concept of an *abstract syntax tree* to represent a program in an object language, and we also use the same concept to talk about meta-language programs. The nodes of such a tree are expressions, and conversely, the children of a node are the sub-expressions. We will also refer to sub-expressions as *branches*.

In the abstract syntax representation of programs, we refer to the names of variables in an object language as *atoms*. An atom is any occurrence of such a name, be it in a declaration (which we also call *binding occurrence*) or a reference. An important point of discussion will be reasoning about sets of different kinds of *free atoms* in the terms of an object language program.

2.2 Syntax

Figure 2.1 shows the syntax of Romeo. A Romeo program is a collection of recursively defined functions that operate on typed terms, and an expression to evaluate. The latter is not important for our presentation. We will focus on the function definitions.

An important observation is that Romeo only allows variables x in many places where one would usually expect expressions. We marked these occurrences with the superscript e (that means, x^e is the occurrence of a variable where one would expect an

expression). This means that all occurrences of x^e are references in the meta-language, whereas the occurrences of x that are not marked are binding occurrences in the meta-language, except one: the single x in e^{qlit}, which is the one place where on would normally expect a variable in a reference position. This restriction has a reason that we will discuss in this chapter; partially lifting it is part of our contribution and will be discussed in section 4.1.

A function definition consists of a name f, a declaration of several arguments and their types $\overline{x{:}\,\tau}$, a precondition C_{pre}, a return type τ, an expression as the function body e and a postcondition C_{post}.

Romeo's type system is used to specify terms of object languages. It knows two different types of atoms as base types: Binder and Ref (short for reference). As the names suggest, Binder is the type for atoms in binding occurrences, whereas Ref is the type for atoms in reference positions. Values of both are drawn from the same countably infinite set of names. Furthermore, there are binary sums of the form $(\tau_0 + \tau_1)$, recursive types of the form $\mu X.\,\tau$,

$$
\begin{array}{rcl}
fD \in FnDef & ::= & (\textbf{define-fn}\ \big(f\ \overline{x{:}\,\tau}\ \textbf{pre}\ C_{pre}\big){:}\ \tau\ e\ \textbf{post}\ C_{post}) \\
e \in Expr & ::= & e^{qlit} \mid (\textbf{call}\ f\ \overline{x^e}) \mid (\textbf{case}\ x^e\ (x_0\ e_0)\ (x_1\ e_1)) \mid \\
 & & (\textbf{open}\ x^e\ \textbf{as}\ \bar{x}\ \textbf{in}\ e) \mid (\textbf{fresh}\ x\ \textbf{in}\ e) \mid \\
 & & (\textbf{let}\ x\ \textbf{be}\ e_{val}\ \textbf{where}\ C\ \textbf{in}\ e_{body}) \mid \\
 & & (\textbf{if}\ x^e_l\ \textbf{equals}\ x^e_r\ \textbf{then}\ e_t\ \textbf{else}\ e_e) \\
e^{qlit} \in QuasiLit & ::= & x \mid (\textbf{ref}\ x^e) \mid \big(\textbf{inj}_0\ e^{qlit}\ \tau_1\big) \mid \big(\textbf{inj}_1\ \tau_0\ e^{qlit}\big) \mid \\
 & & (\textbf{prod}\ \overline{[e^{qlit} \downarrow \beta]} \Uparrow \beta) \\
\tau \in Type & ::= & \text{Binder} \mid \text{Ref} \mid (\tau_0 + \tau_1) \mid \mu X.\,\tau \mid \text{Prod}\ \overline{[\tau \downarrow \beta]} \Uparrow \beta \\
C \in Constraint & ::= & \text{true} \mid z =_{\text{val}} e^{qlit} \mid s = s \mid s \neq s \mid s \# s \mid s \sqsubseteq s \mid C \wedge C \\
s \in SetDesc & ::= & \emptyset \mid sf(z) \mid \mathcal{F}_e(\Gamma) \mid (s) \mid s \cup s \mid s \cap s \mid s \setminus s \\
sf \in SetFn & ::= & \mathcal{F} \mid \mathcal{F}_b \mid \mathcal{F}_r \mid \mathcal{F}_x \\
v \in Value & ::= & \textbf{vatom}\ a \mid \textbf{inj}_0\ v \mid \textbf{inj}_1\ v \mid \textbf{prod}\ \bar{v} \\
z \in SetVar & ::= & x \mid \cdot \\
\Gamma \in Type\text{-}Env & ::= & \overline{z{:}\,\tau} \\
\rho \in Val\text{-}Env & ::= & \overline{z{:}\,v} \\
\beta \in Bindspec & ::= & \bar{\ell} \\
a \in Atom & & \\
\ell \in Number & &
\end{array}
$$

Figure 2.1 – Syntax of Romeo

and n-ary products of the form (**prod** $\overline{[e^{qlit} \downarrow \beta]} \Uparrow \beta$). While sums and recursive types have the usual form[1], product types contain what we will call *binding specifications*, namely an *import* specification $\downarrow \beta$ for each field, and an *export* specification $\Uparrow \beta$ for the whole product. Their meaning will be discussed in the next section.

[1] Pierce [1] gives a very good introduction to the standard parts of the type systems that we use here

An expression e is either a *quasi-literal* e^{qlit}, a function call (**call** $f\ \bar{x}$), or a `case`-, `open`-, `fresh`-, `let`-or `if`-expression.

A quasi-literal is either a variable (reference) x, a binder-to-reference conversion (**ref** x^e) or a constructor. In (**ref** x^e), x^e must have type Binder, the expression will then return an atom of type Ref. The constructors $\mathbf{inj_0}$ and $\mathbf{inj_1}$ are the introduction form for values of sum types—they take a value and the type of the other case of the sum. The constructor **prod** constructs a value of a product type, the binding specifications are the same as in the formulation product types. Note that constructors can be nested.

A function call to a function f comes with some variables $\bar{x}$ as arguments.

A `case`-expression determines the option of the value stored in x and binds it to the corresponding x_i in the corresponding e_i.

An `open`-expression takes a value of a product type stored in the variable x and binds its fields to the variables $\bar{x}$ in the expression e.

A `fresh`-expression introduces a new variable x that stores a *fresh* atom of type Binder. The fact that the atom is fresh means that its name is guaranteed to be distinct from all the names that are already in use.

A `let`-expression declares a new variable x and binds it to the value of first sub-expression e_{val}, which we will call the *value expression*. This value must satisfy the constraint C, which for convenience we also call the *postcondition* of the let-expression (although it is only a postcondition on the value expression). The variable x is then available in the *body expression* of the `let`, e_{body}.

An `if`-expression compares two atoms (that is, x_l^e and x_r^e must either both have type Binder or both have type Ref), and depending on their equality, either the *then-expression* e_t or the *else-expression* e_e is evaluated.

Constraints can be either pre- or postconditions of functions or postconditions on values bound by a let-expression. They describe relations between sets of atoms in terms that must hold at certain points in the program. The proof system that we will see later is used to statically determine that that is actually the case. A constraint is either true, an $=_{\text{val}}$-constraint, one of four relations (equality $=$, inequality $\neq$, disjointness #, inclusion $\subseteq$) between sets or the $\wedge$-connective of two constraints. Constraints of the form $z \ =_{\text{val}} e^{qlit}$ are a shorthand to express that z was constructed using values from other variables. This constraint cannot be annotated by a user of Romeo, but is used by the proof system (and subsequently eliminated in a simplification pass). Set expressions can be the empty set $\emptyset$, the standard set operations $\cup$, $\cap$ and $\setminus$ (the parentheses are there to control precedence) and functions on variables producing sets of atoms.

We care about the following sets of atoms: the free atoms $\mathcal{F}$, the free binders $\mathcal{F}_b$, the free references $\mathcal{F}_r$, or the exposable atoms $\mathcal{F}_x$. We will define their exact meaning in later sections, for now, we only give an intuition of $\mathcal{F}_r$ — $\mathcal{F}_r(z)$ is the set of the free (i.e. unbound) references in the value represented by the variable z.

The symbol $\cdot$ is a special variable that represents the result of a function or expression in a postcondition. It should not appear in a precondition written by a programmer because the result is not available at the time of a function call, which is when the precondition must be checked.

Run-time values v can be either atoms a, sum-values tagged with $\mathbf{inj}_0$ or $\mathbf{inj}_1$, or records of a product type with field values $\bar{v}$. Value environments ρ keep track of the values of variables at run-time, while type-environments Γ keep track of their types (both at run-time and during checking).

2.3 Binding

The core part of Romeo is its system to describe the binding structure of terms. A term is either an atom, corresponding to a leaf in an abstract syntax tree, or an arbitrary-size tuple of terms, corresponding to a node. Atoms may be binders that introduce bindings, or references that may be bound. The import- and export-specifications of product types specify scoping rules: what is imported and exported are bindings. A binder by definition exports a binding. That alone does not bind any variables—the binding needs to be imported somewhere. An import specification in a product means that the sub-term that it belongs to is in the scope of the declarations that generate the bindings that are imported. Those imported bindings are specified by natural numbers, which refer to the exported bindings of the sub-terms with the respective indices. Similarly, an export specification in a product propagates the exports of the sub-terms with the respective indices.

2.3.1 Binding Specifications

We will demonstrate the actual meaning of the binding specifications with the following examples.

$$\text{Prod}\ ([\text{Binder} \downarrow \emptyset]\ [E \downarrow (0)]) \Uparrow \emptyset$$

Example 2.1 – A type for a λ-abstraction

Example 2.1 shows the basic idea of importing bindings within a product. It represents a λ-abstraction ($\lambda x. e$) with its two components: a body expression e and a name x that is bound in that body. The type of the binding occurrence of x is `Binder`. For the body, we assume that E is some type that represents expressions e in the object language. The first field of the product does not import anything, but the second field (the body) imports all bindings from the first (hence the index 0). This means that the body is in the scope of the declaration of x, so all occurences of x in e that have type `Ref` are bound. Nothing outside of the λ-abstraction can be in scope of either the declared variable or anything in the body, so we export nothing.

$$\mu E. (\text{Ref} + (\text{Prod}([\text{Binder}{\downarrow}\emptyset]\ [E \downarrow (0)]) \Uparrow \emptyset + \text{Prod}([E]\ [E]) \Uparrow \emptyset))$$

Example 2.2 – Specifying λ-terms

We can extend the previous example a bit to complete a basic representation of λ-terms. Example 2.2 shows a full specification. A λ-term is either a variable reference, a λ-abstraction (which we already saw in Example 2.1, although now the type variable E is bound), or an

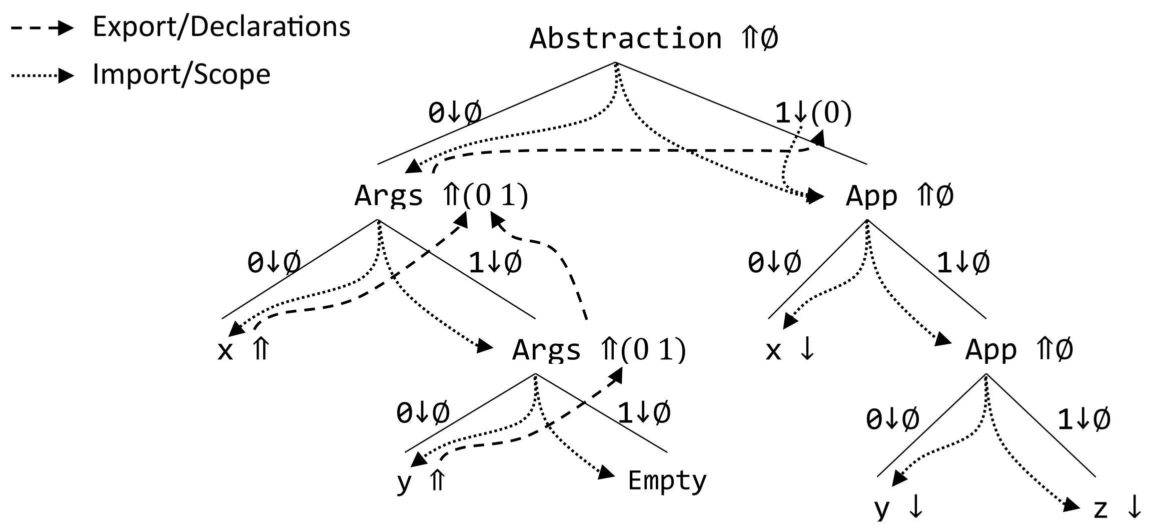


Figure 2.2 – The flow of bindings for the λ-term $\lambda x\ y.\left(x\ (y\ z)\right)$

application, which is a pair of two λ-terms. No additional imports or exports are specified, since abstractions are the only places where binders are introduced.

If we extend our language slightly to allow λ-abstractions with an arbitrary number of declarations, we also get to see a non-empty export specification.

$$\text{Prod}\ ([\ \mu A.\,(\text{Prod}() \Uparrow \emptyset + \text{Prod}([\text{Binder}]\ [A]) \Uparrow (0\ 1))\ \ ]\ \ [E \downarrow (0)]) \Uparrow \emptyset$$

Example 2.3 – Argument list for λ-abstraction with multiple arguments

In Example 2.3, we specify a list of multiple argument declarations. The non-empty product can be seen as describing a cons-cell that exports both the values exported from its head and its tail. If we replace the simple `Binder` from Example 2.1 with this argument list specification, we get the specification of a λ-abstraction with multiple arguments.

Figure 2.2 shows an example of how the binding specifications in the above examples work out for the λ-term $\lambda x\ y.\left(x\ (y\ z)\right)$. The left-hand side is a list of binders, in this case containing x and y. All binders are exported by the left-hand side (`Empty` stands for the empty product at the end of the list and of course does not export anything), and imported by the right-hand side of the abstraction. There, they join all the other variables that already may be in scope (represented by the dotted arrows coming down from the `Abstraction` node)—every node propagates the declarations already in scope. This way, x and y in the body of the λ-abstraction get correctly bound to the declarations on the left-hand side, while z is free, but may be bound by some declaration outside of the abstraction.

To get an idea of how the internal representation of object language terms in Romeo works, Example 2.4 shows the representation of $\lambda x\ y.\,(x\ (y\ z))$ in a structure of the type given by combining Example 2.2 and Example 2.3.

$$(\mathbf{inj}_1\ (\mathbf{inj}_0\ (\mathbf{prod}\ (\mathbf{prod}(\mathbf{vatom}\ x,\mathbf{prod}(\mathbf{vatom}\ y,\mathbf{prod}()))),$$
$$(\mathbf{prod}(\mathbf{vatom}\ x,(\mathbf{prod}\ (\mathbf{vatom}\ y,\ \mathbf{vatom}\ z)))))))$$

Example 2.4 – Representing $(\lambda\ (x\ y)\ (x\ (y\ z)))$ as Romeo value

It should be noted that the binding specifications are presented here in a slightly simplified way. The simplification is that we describe β as a list of indices. In actual Romeo, βs are indices that are combined by different operators which describe what happens if the same name is encountered twice.

2.3.2 Free Atoms

Now that we have discussed how binding specifications work in Romeo, we can explain the meaning of the set functions $\mathcal{F}, \mathcal{F}_b, \mathcal{F}_r$, and $\mathcal{F}_x$. The set functions $\mathcal{F}_b$ and $\mathcal{F}_r$ are the most important, because the other two can be expressed in terms of them.

The *free binders* $\mathcal{F}_b$ are the exported declarations of a term and the *free references* $\mathcal{F}_r$ are its free variables (we already stated that in section 2.2). The *free atoms* $\mathcal{F}$ in a term are defined as the union of the former two. The *free atoms in the environment* $\mathcal{F}_\mathrm{e}$ are defined as the union of the free atoms of all terms represented by variables in the scope of the given environment Γ, and the *exposable atoms* $\mathcal{F}_x$ are the free binders in the sub-expressions of a product that are not exported (such as the x in $\lambda x. e$). The latter are named in a slightly odd way, since the exposable atoms are atoms that may *not* escape. The reason for that is that those names are local to their expression (again, think of the x in $\lambda x. e$). They should not be able to bind anything outside of their expression, nor should they be a free name floating around outside of where they ought to be.

2.4 The Type System

The typing rules are very simple. Variables in Romeo may not be shadowed—whenever a variable is added to the environment, it must be unique. This makes some things easier, as we will see in section 2.7. There is no polymorphism and no subtyping. Recursive types are equi-recursive. Functions are annotated with their argument and return types, the type of their body expression must exactly match their declared return type, and the types of arguments in function calls must exactly match the annotated argument types. Similarly, the types of atoms in the comparison of an if-expression must match.

2.5 Semantic Subtleties

The semantics of Romeo, given by its *execution system*, are largely what one would expect from the constructs we have in the syntax—`if` checks for equality of two atoms and then continues evaluation with either the `then`- or the `else`-branch depending on the result of that check, `let` evaluates its value expression and then its body with the value available in the variable introduced by `let`, and so on. However, `open` is a bit special. It uses the binding information in the types to ensure that name clashes will not occur. Therefore, types cannot be erased, but have to be kept around during execution.

As an example of what open does, consider the aliasing problem discussed by Herman [7]. Example 2.5 shows the macro `first-arg`. It takes two names and uses them to generate two nested λ-abstraction, creating a curried binary function that returns its first argument. Note that in this macro, the names of the variables are determined by the user of the macro. That is, in Romeo, both arguments to the macro would have type Binder.

```
(define-syntax first-arg
 (syntax-rules ()
   [(first-arg x y) (lambda (x) (lambda (y) x))]))


> (((first-arg a a) 5) 3)

➔ (((lambda (a) (lambda (a) a)) 5) 3)

➔ ((lambda (a) a) 3)

➔ 3
```

Example 2.5 – The aliasing problem

Example 2.5 also shows user input that that can make naïve macro expansion fail to preserve the intended binding structure of the program. When it is given the same name twice, applying the resulting function to two values will return the second value instead of the first. Example 2.6 shows a Romeo program with simplified syntax (we omitted the constraints in the function declaration, and the italic parts are either shorthands for types (*first-arg-type*, *λ-term-type*) or a function *mk-lam* that produces a quasi-literal representing a λ-abstraction, of the type we showed in Example 2.1).

```
first-arg-type = Prod([Binder][Binder])

(define-fn (t-first-arg ([farg first-arg-type])) : λ-term-type
  (open farg as (x y) in
    (mk-Lam x (mk-Lam y (ref x)))))
```

Example 2.6 – first-arg in Romeo (simplified)

The really important part is the open-expression. It traverses the term stored in `farg` and uses the type information—in this case we know that `farg` is a product with two binders—to α-rename atoms that might otherwise cause name clashes. In case of the example user

$$\frac{\rho(x^e_{obj}) = inj_0\ v_0 \quad \Gamma(x^e_{obj}) = (\tau_0 + \tau_1) \quad \Gamma, x_0{:}\tau_0 \vdash_{exe} \langle e_0, \rho[x_0 \rightarrow v_0]\rangle \overset{k}{\Rightarrow} v}{\Gamma \vdash_{\text{exe}} \langle(\textbf{case}\ x^e_{obj}\ (x_0\ e_0)\ (x_1\ e_1)), \rho\rangle \overset{k+1}{\Longrightarrow} v} \text{ E-CASE-LEFT}$$

$$\frac{\begin{array}{c} a \notin \text{fa-env}(\Gamma, \rho) \\ \Gamma, x{:}\text{Binder} \vdash_{\text{exe}} \langle e, \rho[x \rightarrow a]\rangle \overset{k}{\Rightarrow} v \quad \Gamma, x{:}\text{Binder} \vdash_{\text{type}} e{:}\tau \quad v = \text{FAULT} \lor a \notin \text{fa}(\tau, v)\end{array}}{\Gamma \vdash_{\text{exe}} \langle(\textbf{fresh}\ x\ \textbf{in}\ e), \rho\rangle \overset{k+1}{\Longrightarrow} v} \text{ E-FRESH-OK}$$

$$\frac{\begin{array}{c} a \notin \text{fa-env}(\Gamma, \rho) \\ \Gamma, x{:}\text{Binder} \vdash_{\text{exe}} \langle e, \rho[x \rightarrow a]\rangle \overset{k}{\Rightarrow} v \quad \Gamma, x{:}\text{Binder} \vdash_{\text{type}} e{:}\tau \quad v \neq \text{FAULT} \land a \in \text{fa}(\tau, v)\end{array}}{\Gamma \vdash_{\text{exe}} \langle(\textbf{fresh}\ x\ \textbf{in}\ e), \rho\rangle \overset{k+1}{\Longrightarrow} \text{FAULT}} \text{ E-FRESH-ESCAPE}$$

Figure 2.3 – Some execution rules of Romeo

input above, open would rename one of the two identifiers, and we would get the correct result again.

Figure 2.3 shows a selection of Romeo's big-step execution rules. E-CASE-LEFT is a simple first example. When we encounter a `case`-expression, we check whether the value of the given variable x^e_{obj} is tagged with $\mathbf{inj_0}$. If it is, we look up its type τ_0, bind the inner value v_0 of the tagged sum to the variable x_0 in the expression e_0 and evaluate e_0 with an updated type environment Γ and variable environment ρ.

The rules for `fresh` are a bit more complex. The important part is the condition on the right—this is also where the difference between the OK- and the FAULT-rule is. In order for a `fresh`-expression not to fault, the atom a that it generated may not be free in the result. The reason why it may not be free is that it might stay free. Then, if there are two α-equivalent programs, and we add distinct fresh and free variables to each of them, they are not α-equivalent any more.

```
(define-fn (maybe-faulty (x λ-abs-type)) : some-type
  …)
```

Example 2.7 – Skeleton for examples of faulty and non-faulty programs

Example 2.7 shows the skeleton of a Romeo function called `maybe-faulty`. We again use a slightly simplified syntax for clarity—the code in the example specifies `maybe-faulty` as a function that takes a term representing a λ-abstraction and returns a term of some type that will depend on the example. The body of the function consists of ellipses, and we will now fill in several expressions there and see what is faulty and what is not. Imagine that maybe-faulty is called with some arbitrary name in the object language program as argument.

1.) x – valid. Identity functions are not a problem for hygiene – clearly, α-equivalent inputs are mapped to α-equivalent outputs.

2.) (fresh y in x) – valid. The fresh binder in y is never used and does not escape; `maybe-faulty` is still an identity function.

3.) (fresh y in y) – faulty. The fresh binder in y is exported (because that is what values of type `Binder` do), therefore the result of the function now contains a free binder, therefore a free atom, and that free atom is the atom introduced by fresh, violating the corresponding condition $a \notin \mathrm{fa}(\tau, v)$ in E-FRESH-OK.

4.) (fresh y in (*mk-lam* y (**ref** y))) – valid. All inputs get mapped to an identity function. The actual name in y does not matter – it is confined to the λ-abstraction (because the type of a λ-abstraction does not specify any exports, and the reference to y is bound) and does not appear free in the result.

5.) (**fresh** y **in** (**fresh** z **in** (*mk-lam* y (**ref** z)))) – faulty. The binders in y and z are guaranteed to be different, so the binder in y cannot bind the reference atom created from z. This means that we introduce a free variable into the result, and that free variable is different from any other free variable. Therefore, two α-equivalent inputs will be mapped to functions with different free variables, and thus not be α-equivalent any more. Again, the condition $a \notin \text{fa}(\tau, v)$ would be violated, and thus E-Fresh-Escape will fire instead of E-Fresh-Ok, and execution will fault.

The open-rule is even more complex, which is why we refer to Stansifer [19] for an in-depth discussion. For our purposes, the intuition suffices that it uses the binding information in the types to α-rename terms such that they are fresh relative to the current environment, which will prevent name clashes. This is also called *freshening*, a technique already used by the KFFD algorithm [4]. While KFFD had to fully expand all macros first, because binding information would only be available at that point, the types in Romeo give the open-rule all the information it needs to perform the renaming before expansion.

The behavior of open in turn ensures that `if` is well-behaved with regards to α-equivalence—two atoms are equal iff they are related by binding. We will see in chapter 5 that this is useful when we want to implement capture-avoiding substitution for the λ-calculus.

2.6 The Proof System

The rules of the execution system presented in the previous section ensure that a program is transformed in a hygienic way at the time when the transformation actually happens. But the proof system that we will present in this section enables us to prove statically—that is, before a Romeo program is actually run on some object language program—that the execution system will not fault and hence the program will always preserve α-equivalence.

Figure 2.4 shows the rules of the proof system as given by Stansifer and Wand [19]. It starts at the root of each abstract syntax tree that represents a Romeo function definition (P-PROG). The rule P-FNDEF takes the precondition C_{pre}, the postcondition C_{post} and the body expression e from the function definition and constructs the initial state: $\Gamma \vdash_{proof} \{C_{pre}\}\ e\ \{C_{post}\}$. The type environment Γ is initialized with the formal function arguments and their types $\overline{x{:}\,\tau}$.

From this initial state, we traverse the abstract syntax tree. While doing that, we collect additional pre- and post-conditions for the current branch we are on. At the leaves of the abstract syntax tree, proof obligations are generated from the current set of pre- and post-conditions. We will now explain this in more detail.

In general, the rules of the proof system form four groups:

- P-FRESH and P-OPEN introduce new Romeo variables (x for P-FRESH, $\bar{x}$ for P-OPEN). The expressions that these variables represent may contain some kind of free atoms in the object languages. As we saw in section 2.5, some of these atoms may not escape.

$$\frac{\overline{\vdash fD} \quad \epsilon \vdash_{\text{proof}} \{\text{true}\}\ e\ \{\text{true}\}}{\vdash_{\text{proof}} \overline{fD}\ e\ \text{ok}}\ \textsc{P-Prog}$$

$$\frac{\overline{x{:}\,\tau} \vdash_{\text{proof}} \{C_{pre}\}\ e\ \{C_{post}\}}{\vdash_{\text{proof}} \left(\textbf{define-fn}\ \left(f\ \overline{x{:}\,\tau}\ \textbf{pre}\ C_{pre}\right){:}\,\tau_{ret}\ e\ \textbf{post}\ C_{post}\right)\ \text{ok}}\ \textsc{P-FnDef}$$

$$\frac{x \text{ fresh for } \Gamma, H, P \quad \Gamma, x{:}\text{Binder} \vdash_{\text{proof}} \{H \wedge \mathcal{F}(x) \# \mathcal{F}_e(\Gamma)\}\ e\ \{P \wedge \mathcal{F}(x) \# \mathcal{F}(\cdot)\}}{\Gamma \vdash_{\text{proof}} \{H\}\ (\textbf{fresh}\ x\ \textbf{in}\ e)\ \{P\}}\ \textsc{P-Fresh}$$

$$\frac{\begin{array}{c}\overline{x_{sub}} \text{ fresh for } \Gamma, H, P \quad \Gamma\left(x^e_{obj}\right) = \text{Prod}\overline{[\tau_{sub} \downarrow \beta_{sub}]} \Uparrow \beta_e \\ \Gamma, \overline{x_{sub}{:}\,\tau_{sub}} \vdash_{\text{proof}} \left\{H \wedge \mathcal{F}_x\left(x^e_{obj}\right) \# \mathcal{F}_e(\Gamma) \wedge x^e_{obj} =_{\text{val}} \left(\textbf{prod}\overline{[x_{sub} \downarrow \beta_{sub}]} \Uparrow \beta_e\right)\right\}\ e\ \left\{P \wedge \mathcal{F}_x\left(x^e_{obj}\right) \# \mathcal{F}(\cdot)\right\}\end{array}}{\Gamma \vdash_{\text{proof}} \{H\}\ \left(\textbf{open}\ x^e_{obj}\ \textbf{as}\ \overline{x_{sub}}\ \textbf{in}\ e\right)\ \{P\}}\ \textsc{P-Open}$$

$$\frac{\begin{array}{c}\Gamma(x) = (\tau_0 + \tau_1) \quad x_0, x_1 \text{ fresh for } \Gamma, H, P \\ \Gamma, x_0{:}\,\tau_0 \vdash_{\text{proof}} \{H \wedge x^e =_{\text{val}} (\textbf{inj}_0\ x_0\ \tau_1)\}\ e_0\ \{P\} \quad \Gamma, x_1{:}\,\tau_1 \vdash_{\text{proof}} \{H \wedge x^e =_{\text{val}} (\textbf{inj}_1\ \tau_0\ x_1)\}\ e_1\ \{P\}\end{array}}{\Gamma \vdash_{\text{proof}} \{H\}\left(\textbf{case}\ x^e\ (x_0\ e_0)\ (x_1\ e_1)\right)\{P\}}\ \textsc{P-Case}$$

$$\frac{\Gamma \vdash_{\text{proof}} \{H \wedge \mathcal{F}(x^e_l) = \mathcal{F}(x^e_r)\}\ e_0\ \{P\} \quad \Gamma \vdash_{\text{proof}} \{H \wedge \mathcal{F}(x^e_l) \# \mathcal{F}(x^e_r)\}\ e_1\ \{P\}}{\Gamma \vdash_{\text{proof}} \{H\}\ (\textbf{if}\ x^e_l\ \textbf{equals}\ x^e_r\ \textbf{then}\ e_0\ \textbf{else}\ e_1)\ \{P\}}\ \textsc{P-IfEq}$$

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

$$\frac{\text{typeof}(\Gamma, e^{qlit}) = \tau \quad \Gamma, \cdot{:}\,\tau; H \wedge (\cdot =_{\text{val}} e^{qlit}) \vDash_{\text{hyp}} P}{\Gamma \vdash_{\text{proof}} \{H\}\ e^{qlit}\ \{P\}}\ \textsc{P-QLit}$$

$$\frac{\begin{array}{c}\text{rettype}(f) = \tau \\ \text{formals}(f) = \overline{x_{formal}} \quad \text{argtype}(f) = \overline{\tau_{formal}} \quad \Gamma; H \vDash_{\text{hyp}} \text{pre}(f)\overline{[x^e_{actual}/x_{formal}]} \\ \Gamma, \cdot{:}\,\tau; H \wedge \mathcal{F}(\cdot) \subseteq \mathcal{F}_e(\overline{x^e_{actual}{:}\,\tau_{formal}}) \wedge \text{post}(f)\overline{[x^e_{actual}/x_{formal}]} \vDash_{\text{hyp}} P\end{array}}{\Gamma \vdash_{\text{proof}} \{H\}\ \left(\textbf{call}\ f\ \overline{x^e_{actual}}\right)\ \{P\}}\ \textsc{P-Call}$$

Figure 2.4 – The proof system of Romeo

The proof system rules mirror the execution rules in that they add the postcondition that the free/exposable atoms $\mathcal{F}(x)/\mathcal{F}_x(x^e_{obj})$ may not be free in the result $\mathcal{F}(\cdot)$.

- P-Case and P-IfEq each have two branches. Depending on which branch they take in the execution, either e_0 or e_1 is evaluated, and so the proof system must check that in both cases the postcondition P will be satisfied. In each branch, the knowledge about why a particular branch was taken can be added to the knowledge. For example, in the `then`-branch of an `if`, we know that the two compared atoms must be equal.

- P-LET also has two branches. For the value expression e_{val}, we need to prove that it satisfies the postcondition C. Then we can use this additional knowledge in C that describes the value now in x and add C to the knowledge that we have for the body expression e_{body}. In addition to that, we know that $\mathcal{F}(x) \subseteq \mathcal{F}_e(\Gamma)$, that is, the free atom in the result of the value expression must be a subset of the free atoms in all the variables currently in the environment. If there are additional free variables in the result, they must have escaped from a `fresh`- or `open`-expression, which is illegal and prevented elsewhere.

- P-QLIT and P-CALL. Quasi-literals and function calls are always (and the only possible) leaves of the abstract syntax tree, therefore one of these rules must eventually be encountered on every branch. Here, finally, proof obligations are generated (with $\vDash_{\text{hyp}}$). P-QLIT generates the special $=_{\text{val}}$-constraint to express that the result of the current branch is some quasi-literal. P-CALL generates two proof obligations. The first asserts that the preconditions of the called function are satisfied. The second asserts that the knowledge from the postcondition of the called function combined with the current knowledge suffice to prove our current postcondition.

2.7 Logic

The relation $\vDash_{\text{hyp}}$ is defined in terms of logical entailment $\vDash$ under the rules of Figure 2.5 and the underlying atom functions in Figure 2.6. It relates the proof obligations to the execution system, stating that if we can prove that $\Gamma; H \vDash_{\text{hyp}} P$, then H implies P under all possible value environments (i.e. variable assignments).

Figure 2.6 gives an exact definition of the different sets of atoms that we will consider—free binders ($\mathcal{F}_b$), free references ($\mathcal{F}_r$), free atoms ($\mathcal{F}$), free atoms in the environment ($\mathcal{F}_e$) and exposable atoms ($\mathcal{F}_x$).

Note that the translation of $S[\![\mathcal{F}_e(\Gamma_{subject})]\!]_{\Gamma,\rho}$ to $\texttt{fa-env}(\Gamma_{subject}, \rho)$ requires that $\text{dom}(\Gamma_{subject}) \subseteq \text{dom}(\rho)$ by the definition of `fa-env`. This is always true because Romeo does not allow shadowing of variables, and thus environments can only grow when going down in the abstract syntax tree, and not change otherwise. Proof obligations are generated at the leaves of the abstract syntax tree, and $\mathcal{F}_e(\Gamma_{subject})$ must have been generated on the way. The environment $\Gamma_{subject}$ is therefore a snapshot of the environment Γ taken at some point, which means $\text{dom}(\Gamma_{subject}) \subseteq \text{dom}(\Gamma)$. Since Γ and ρ always have the same domain, the translation is safe.

2.8 Simplification and Approximation

Before the proof obligations generated by the proof system go to the SMT-solver, they are simplified, and the sizes of their sets are approximated.

$$\begin{array}{rcl}
\Gamma; \rho; H \vDash_{\text{hyp}} P & \Leftrightarrow & \text{if } \Gamma; \rho \vDash H, \text{ then } \Gamma; \rho \vDash P \\
\Gamma; H \vDash_{\text{hyp}} P & \Leftrightarrow & \forall \rho. \text{ if } \Gamma \vdash_{\text{type-env}} \rho \text{ and } \Gamma; \rho \vDash H, \text{ then } \Gamma; \rho \vDash P
\end{array}$$

$$\begin{array}{rcl}
\Gamma; \rho \vDash C_0 \wedge C_1 & \Leftrightarrow & \Gamma; \rho \vDash C_0 \text{ and } \Gamma; \rho \vDash C_1 \\
\Gamma; \rho \vDash s_0 = s_1 & \Leftrightarrow & S[\![s_0]\!]_{\Gamma,\rho} \text{ equals } S[\![s_1]\!]_{\Gamma,\rho} \\
\Gamma; \rho \vDash s_0 \# s_1 & \Leftrightarrow & \Gamma; \rho \vDash s_0 \cap s_1 = \emptyset \\
\Gamma; \rho \vDash s_0 \subseteq s_1 & \Leftrightarrow & S[\![s_0]\!]_{\Gamma,\rho} \subseteq S[\![s_1]\!]_{\Gamma,\rho} \\
\Gamma; \rho \vDash z =_{\text{val}} e^{qlit} & \Leftrightarrow & Q[\![z]\!]_{\rho} \text{ equals } Q[\![e^{qlit}]\!]_{\rho} \\
\Gamma; \rho \vDash \textbf{true} & \Leftrightarrow & \text{Always}
\end{array}$$

$$\begin{array}{rcl}
Q[\![\_]\!]_{\_} & : & QuasiLit \times Type\text{-}Env \rightarrow Value \\
Q[\![x]\!]_{\rho} & = & \rho(x) \\
Q[\![(\textbf{ref}\, x^{e})]\!]_{\rho} & = & \rho(x) \\
Q[\![(\textbf{inj}_{\mathbf{0}}\ e^{qlit}\ \tau)]\!]_{\rho} & = & \textbf{inj}_{\mathbf{0}}\ Q[\![e^{qlit}]\!]_{\rho} \\
Q[\![(\textbf{inj}_{\mathbf{1}}\ \tau\ e^{qlit})]\!]_{\rho} & = & \textbf{inj}_{\mathbf{1}}\ Q[\![e^{qlit}]\!]_{\rho} \\
Q[\![(\textbf{prod}\ \overline{e^{qlit} \downarrow \beta_{im}}\ \Uparrow \beta_{ex})]\!]_{\rho} & = & \textbf{prod}\ \overline{Q[\![e^{qlit}]\!]_{\rho}}
\end{array}$$

$$\begin{array}{rcl}
S[\![\_]\!]_{\_,\_} & : & SetDesc \times Type\text{-}Env \times Val\text{-}Env \rightarrow \mathbb{P}(Atom) \\
S[\![\emptyset]\!]_{\Gamma,\rho} & = & \emptyset \\
S[\![s_0 \cup s_1]\!]_{\Gamma,\rho} & = & S[\![s_0]\!]_{\Gamma,\rho} \cup S[\![s_1]\!]_{\Gamma,\rho} \\
S[\![s_0 \cap s_1]\!]_{\Gamma,\rho} & = & S[\![s_0]\!]_{\Gamma,\rho} \cap S[\![s_1]\!]_{\Gamma,\rho} \\
S[\![\mathcal{F}_e(\Gamma_{subject})]\!]_{\Gamma,\rho} & = & \texttt{fa-env}(\Gamma_{subject}, \rho) \\
S[\![\mathcal{F}_b(z)]\!]_{\Gamma,\rho} & = & \texttt{fb}(\Gamma(z), \rho(z)) \\
S[\![\mathcal{F}_r(z)]\!]_{\Gamma,\rho} & = & \texttt{fr}(\Gamma(z), \rho(z)) \\
S[\![\mathcal{F}(z)]\!]_{\Gamma,\rho} & = & \texttt{fa}(\Gamma(z), \rho(z)) \\
S[\![\mathcal{F}_x(z)]\!]_{\Gamma,\rho} & = & \texttt{fx}(\Gamma(z), \rho(z))
\end{array}$$

Figure 2.5 – The interpretation of $\vDash_{\text{hyp}}$

$B[\![\beta]\!]$ … function that takes a list of sets and returns the union of the sets at indices β

$$\begin{aligned}
\mathtt{fb}(\_,\_) &\quad : \quad \tau \times v \rightarrow \bar{a} \\
\mathtt{fb}(\mathbf{Prod}\,\overline{\tau \downarrow \beta} \Uparrow \beta_{ex}, \mathbf{prod}\,\overline{v}) &\quad ::= \quad B[\![\beta_{ex}]\!](\overline{\mathtt{fb}(\tau, v)}) \\
\mathtt{fb}(\tau_0 + \tau_1, \mathbf{inj_0}\, v) &\quad ::= \quad \mathtt{fb}(\tau_0, v) \\
\mathrm{fb}(\tau_0 + \tau_1, \mathbf{inj_1}\, v) &\quad ::= \quad \mathtt{fb}(\tau_1, v) \\
\mathtt{fb}(\mu X.\tau, v) &\quad ::= \quad \mathtt{fb}(\tau[\mu X.\tau/X], v) \\
\mathtt{fb}(\mathrm{Binder}, a) &\quad ::= \quad \{a\} \\
\mathtt{fb}(\mathrm{Ref}, a) &\quad ::= \quad \emptyset
\end{aligned}$$

$$\begin{aligned}
\mathtt{fr}(\_,\_) &\quad : \quad \tau \times v \rightarrow \bar{a} \\
\mathtt{fr}(\mathbf{Prod}\,\overline{\tau \downarrow \beta} \Uparrow \beta_{ex}, \mathbf{prod}\,\overline{v}) &\quad ::= \quad \cup\, \overline{fr(\tau, v) \setminus B[\![\beta]\!](\overline{fb(\tau, v)})} \\
\mathtt{fr}(\tau_0 + \tau_1, \mathbf{inj_0}\, v) &\quad ::= \quad \mathtt{fb}(\tau_0, v) \\
\mathtt{fr}(\tau_0 + \tau_1, \mathbf{inj_1}\, v) &\quad ::= \quad \mathtt{fb}(\tau_1, v) \\
\mathtt{fr}(\mu X.\tau, v) &\quad ::= \quad \mathtt{fb}(\tau[\mu X.\tau/X], v) \\
\mathtt{fr}(\mathrm{Binder}, a) &\quad ::= \quad \emptyset \\
\mathtt{fr}(\mathrm{Ref}, a) &\quad ::= \quad \{a\}
\end{aligned}$$

$$\begin{aligned}
\mathtt{fa\text{-}env}(\_,\_) &\quad : \quad \Gamma \times \rho \rightarrow \bar{a} \\
\mathtt{fa\text{-}env}(\Gamma, \rho) &\quad ::= \quad \bigcup_{x \in \Gamma} fa\big(\Gamma(x), \rho(x)\big)
\end{aligned}$$

$$\begin{aligned}
\mathtt{fa}(\_,\_) &\quad : \quad \tau \times v \rightarrow \bar{a} \\
\mathtt{fa}(\tau, v) &\quad ::= \quad \mathtt{fr}(\tau, v) \cup \mathtt{fb}(\tau, v)
\end{aligned}$$

$$\begin{aligned}
\mathtt{fx}(\_,\_) &\quad : \quad \tau \times v \rightarrow \bar{a} \\
\mathtt{fx}(\mathbf{Prod}\,\overline{\tau \downarrow \beta} \Uparrow \beta_{ex}, \mathbf{prod}\,\overline{v}) &\quad ::= \quad \big(\cup\, \mathtt{fb}(\tau, v)\big) \setminus \mathtt{fb}\big(\mathbf{Prod}\,\overline{\tau \downarrow \beta} \Uparrow \beta_{ex}, \mathbf{prod}\,\overline{v}\big)
\end{aligned}$$

Figure 2.6 – Atom functions

Simplification expands $\mathcal{F}_e, \mathcal{F}_a$ and $\mathcal{F}_x$ to their meanings in terms of $\mathcal{F}_b$ and $\mathcal{F}_r$ analogous to the definitions of fa-env, fa and fx in terms of fb and fr. It also removes constraints of the form $z =_{val} e^{qlit}$, and for each of them replaces all occurrences of $\mathcal{F}_{r/b}(z)$ with the knowledge we get from deconstructing the quasi-literal. This works just like partially applying the functions `fb` and `fa`, where we do not have all the second arguments yet (they may be variables, at which point we leave it at $\mathcal{F}_{r/b}(x)$ for each such variable x). For example, for the constraint $z =_{\text{val}} e^{qlit}$, if e^{qlit} constructs some value that does not export anything, we can replace all occurrences $\mathcal{F}_b(z)$ with $\emptyset$. On the other hand, if

$$e^{qlit} = (\textbf{prod}\ ([x \downarrow \emptyset][y \downarrow \emptyset]) \Uparrow \emptyset),$$

that is, it constructs the representation of the application $(x\ y)$ in the λ-calculus, then we know that $\mathcal{F}_r(z) = \mathcal{F}_r(x) \cup \mathcal{F}_r(y)$, i.e. the free variables in the application are union of the free variables in its two sub-expressions. Therefore, we can replace all occurrences of $\mathcal{F}_r(z)$ with $\mathcal{F}_r(x) \cup \mathcal{F}_r(y)$.

The SMT-solver will have to reason over all possible variable environments ρ. As there are infinitely many of them, Romeo uses an approximation: the sets of free binders and references become uninterpreted, atomic sets. This approximation is sound, but not complete, as set cardinalities may play a role that the proof obligations do not express. Romeo employs an *abstract counting* [20] scheme, trying to infer set size bounds of either `zero`, `one`, `zero-or-one` or `one+` (non-zero). For each proof obligation, these cardinality constraints are added to the respective hypotheses for each variable (if they can be inferred). We will further discuss incompleteness in chapter 5.

2.9 Relation to pure FreshML

Romeo's proof system is largely based on that of pure FreshML [18]. Many of the rules are similar, some even identical. However, there are a few differences. Pure FreshML has a more complicated semantics, in particular, there are two sets of values, one of which treats bound variables as bound, whereas the other does not. Romeo's binding specifications are a bit more expressive than pure FreshML's—it is not possible to specify the binding structure of `let*` in pure FreshML. This is because pure FreshML uses the binding specifications of Cαml [21]. Cαml-style "abstractions" do not support nesting in the way that is necessary to specify `let*`, because nested abstraction patterns are conflated into one larger pattern, effectively producing `letrec`.

Another small difference is the approximation scheme for set cardinalities: pure FreshML only distinguishes empty and non-empty sets, while Romeo also knows sets of size one. It turns out that this additional precision yields great benefits, because in many situations, larger finite set sizes can be put together by adding up size-one-constraints.

Finally, there is no formal proof that pure FreshML actually preserves α-equivalence between arbitrary input programs. This proof is currently work-in-progress for Romeo.

Apart from its close relation to Romeo, the main reason why pure FreshML is important to us is that there is a prototype implementation of it by Pottier. We modeled Romeo-L, our language on top of Romeo, after this prototype implementation. This way, we could use examples that were published for pure FreshML to test our own system. We will present the results of these tests in chapter 5.

2.10 Summary and Thesis

We have introduced the proof system of Romeo, designed to statically prove that macros do not let atoms escape the context that they should not escape (and some more user-annotated assertions if required). In the next section, we will see some observations that we make about Romeo, which will support our thesis.

Thesis

Binding hygiene can be checked statically with few annotations and with meaningful explanations for errors.

3 Observations about Romeo

This chapter discusses observations we made about Romeo. These observations are the basis for Romeo-L, a prototype implementation (discussed in chapter 4) of a language based on Romeo.

3.1 Converting proof obligations to SMT input is simple

Since pure FreshML used a SAT solver to discharge proof obligations for static checking, the proof system of Romeo was already designed with SMT solvers in mind. SMT stands for "satisfiable modulo theories", and the basic idea is to have a front-end for SAT solvers that understands more high-level constructs than just Boolean connectives. SMT solvers are also used in other static software verification projects, notably Boogie [22]. In contrast to those wide-purpose-tools, the proof obligations of Romeo have a very limited structure, and we will see in what ways we can exploit that.

3.1.1 Translating Constraints

It turns out that the approximated proof obligations map into a decidable (although NP-complete) fragment of SMT problems. The representation is based on arrays, which again are based on uninterpreted functions. An overview of decidability of array problems is given by Bradley et al. [23]. The concrete theory that we are using here is called *combinatory array logic* (CAL). Its implementation and application to set theory is discussed by de Moura et al. [24]. In SMT-solver terminology, types are called sorts. An array maps elements of one sort

Constraint		**CAL**
$\emptyset$	$\equiv$	$\mathrm{K}_{Atom}(\text{false})$
$\cup$	$\equiv$	map $\vee$
$\cap$	$\equiv$	map $\wedge$
$\setminus$	$\equiv$	$\cap$ (map $\neg$)
$\subseteq$	$\equiv$	(map $\rightarrow$) $=$ $\mathrm{K}_{Atom}(\text{true})$
size-zero	$\equiv$	$= \emptyset$
size-one	$\equiv$	$\exists e.\,(s = \text{store}(\emptyset, e, \text{true}))$
size-zero-or-one	$\equiv$	size-one $\vee$ size-zero
size-one+	$\equiv$	$\neq \emptyset$

Figure 3.1 – Mapping constraints to CAL

to elements of another sort, that is, it is an interpreted function from the first to the second sort. There are four combinators that we care about:

- select(a, i) – returns the value of the array a at index i
- store(a, i, v) – returns a new array that is the same as a, except that value v is stored at index i
- $\mathrm{K}_S(v)$ – returns a new array that maps all indices i of sort S to v
- $map(f, a_1 \ldots a_n)$ – for an n-ary function f, returns a new array a s.t. for every index i, the following holds: $select(a, i) = f(select(a_1, i) \ldots select(a_n, i))$

The theory is extensional which means that two arrays are treated as equal iff they map the same indices to the same values.

Figure 3.1 shows how constraints map to CAL. We introduce an uninterpreted sort *Atom*, and define a sort *Set* to be an array sort from *Atom* to the predefined sort *Bool*. That means that a *Set* is a function that takes an *Atom* as argument and returns whether that *Atom* is an element of the set. An $\wedge$-connective can be expressed by several consecutive assertions—one for each clause.

The empty set is an array that maps every index to false. Union is the pointwise $\vee$ of two arrays, similarly, intersection is expressed by pointwise $\wedge$. Set difference is the intersection of the first set and the complement of the second. Finally, because of the simplification process discussed in section 2.8, the only sets that actually appear are of the form $\mathcal{F}_r(z)$ or $\mathcal{F}_b(z)$. They will be converted into atomic, numbered set constants set_i.

Equality and inequality can be mapped to equality and negated equality. The relation $\subseteq$ can be expressed by mapping pointwise implication on two sets and comparing the result to an array that only contains true (the implication must hold for every index). As in the logic in section 2.7, set disjointness is an assertion that the intersection of two sets is empty. A set has size zero if it is equal to the empty set, and size one if exactly one element is stored into the empty set. Size zero-or-one is the only point where we produce disjunctions. If a set contains at least one element, this can be expressed by asserting that the set is not empty.

This covers all the forms that constraints can have after approximation. The only quantifier that appears here is an existential quantifier for the size-one constraint, which will be skolemized away.

3.1.2 Proof Obligations

Recall that a proof obligation looks like this:

$$\Gamma;\ H \vDash_{\text{hyp}} P$$

Ignoring the environment Γ (which, because we eliminated all occurrences of $\mathcal{F}_e$ in the simplifications step, only serves as a link to the logic defined in section 2.7), we have two sets of constraints. We call H the *hypothesis* and P the *goal*, and want to prove $H \rightarrow P$, or written differently, $\neg H \vee P$. This formula must be valid, so we will ask the SMT-solver whether $H \wedge \neg P$ is satisfiable—if it is, we can extract a model which will serve as a counterexample.

Both H and P consist of a number of $\wedge$-connected clauses. For H, that is not a problem—we just assert each clause individually. $P = P_1 \wedge P_2 \wedge \ldots \wedge P_n$ on the other hand is negated, so by DeMorgan's laws our formula looks like this: $H \wedge (\neg P_1 \vee \neg P_2 \vee \ldots \vee \neg P_3)$. This means that in order to show that $\neg H \vee P$ is valid, we need to show that each of the formulae $H \wedge \neg P_1$, $H \wedge \neg P_2 \ldots H \wedge \neg P_n$ is unsatisfiable. As we will see in sections 3.3 and 4.2, this makes it easier to locate errors while not incurring major overhead.

3.1.3 Decidability and Incompleteness

Because the theory is decidable, an SMT solver should be able to return either `sat` or `unsat`. However, like in pure FreshML [18], the system is incomplete because the sets are atomic, and Romeo does not have cardinality constraints other than the four that were presented.

3.2 Many Assertions are Inferable

Romeo has a very restricted syntax. This makes it easier to prove properties of the language, but also makes it harder to actually write programs in it. Example 3.2 shows a Romeo program. The italic type definitions are shorthands that are not in the syntax. We will briefly explain what the program does.

We consider the core λ-calculus and λ_{let}, which is the λ-calculus extended with a let-expression. These are their possible terms:

$$\begin{aligned} \lambda\text{-term } e &::= x \mid \lambda x.\, e \mid (e\ e) \\ \lambda_{\text{let}}\text{-term } e_{let} &::= x \mid \lambda x.\, e_{let} \mid (e_{let}\ e_{let}) \mid \text{let } \overline{x = e_{let}} \text{ in } e_{let} \end{aligned}$$

Figure 3.2 – Terms of λ and λ_{let}

Their Romeo types are written down in Example 3.2, as the shorthands *lam* and *lam/let*. The function `convert` takes a λ_{let}-term and returns an equivalent λ-term, by converting each let-clause to a λ-expression wrapped in an application. Example 3.1 gives an intuition of how convert is supposed to work.

$$\text{convert}(\text{let } x\ =\ (\lambda y.\, y) \text{ in } (x\ x))\ =\ ((\lambda x.\, (x\ x))\ (\lambda y.\, y))$$

Example 3.1 – How the convert function converts a let-expression of the object language

```
Lam = μL.(Ref + (Prod([Binder ↓∅] [L ↓0])⇑∅ + Prod ([L ↓∅] [L ↓∅])⇑∅))
Lvar = Ref
Labs = Prod([Binder ↓∅] [Lam ↓0]) ⇑∅
Lapp = Prod([Lam ↓∅] [Lam ↓∅])⇑∅

Lam/Let = μL. ((Ref + Prod([Binder ↓∅] [L ↓0]) ⇑∅)
               +
               (Prod ((L ↓∅) (L ↓∅))⇑∅
                +
                Prod ([μLA.(Prod()⇑∅ + Prod([Binder ↓∅] [L ↓∅] [LA ↓∅])⇑(0 2)) ↓∅]
                      [L ↓0]) ⇑∅))
Labs/Let = Prod([Binder ↓∅] [Lam/Let ↓0])⇑∅
Lapp/Let = Prod([Lam/Let ↓∅] [Lam/Let ↓∅])⇑∅

res = (inj₁ Lvar
        (inj₁ Labs (prod ([(inj₁ Lvar (inj₀ (prod ([x ↓∅] [crlet ↓0]) ⇑∅) Lapp)) ↓∅]
                          [cbe ↓∅])⇑∅))))))))))

(define-fn (convert ([e Lam/Let]) pre true): Lam
  (case e
    left (case left
           var (inj₀ var (Labs + Lapp))
           abs (open abs as (abs-b abs-e) in
                 (let ce be (call convert abs-e)
                   where 𝓕r(·) = 𝓕r(abs-e)
                   in (inj₁ Lvar (inj₀ (prod ([abs-b ↓∅] [ce ↓0]) ⇑∅) Lapp))))
    right (case right
            app (open app as (app-l app-r) in
                  (let cl be (call convert app-l)
                    where 𝓕r(·) = 𝓕r(app-l)
                    in (let cr be (call convert app-r)
                         where 𝓕r(·) = 𝓕r(app-r)
                         in (inj₁ Lvar (inj₁ Labs (prod ([cl ↓∅] [cr ↓∅])⇑∅))))))
            let (open let as (let-b let-e) in
                  (case let-b
                    letl (call convert let-e)
                    letr (open letr as (x be rest) in
                           (let rlet be (inj₁ (+ Lvar Labs/Let)
                                          (inj₁ Lapp/Let
                                                (prod ([rest ↓∅][let-e ↓0])⇑∅)))
                             where 𝓕r(·) = 𝓕r(rest) ∪ (𝓕r(let-e)\𝓕b (rest))
                             in (let crlet be (call convert rlet)
                                  where 𝓕r(·) = 𝓕r(rlet)
                                  in (let cbe be (call convert be)
                                       where 𝓕r(·) = 𝓕r(be)
                                       in res
     post 𝓕r(·) = 𝓕r(e))
```

Example 3.2 – Converting let-expressions to λ-expressions in Romeo

There are no preconditions on convert, that is, it can take any λ_{let}-term. But we have a postcondition ($\mathcal{F}r(\cdot) = \mathcal{F}r(e)$, at the end) that asserts that the free references in the output are the same as those in the input, and because neither λ-terms nor λ_{let}-terms export any bindings, we know that the free binders are empty in any case. The function works its way recursively through the given expression and preserves the free variables in the conversion, which is an additional guarantee over those that the proof system gives us automatically.

What we see in Example 3.2 is that there are six let-expressions—they and the postcondition of the function are highlighted to make them easier to find. All of them have to be there because the syntax of Romeo would not allow to nest their value-expression into the expression where they belong. Five of the six let-expressions make a recursive call to `convert`, the sixth contains a quasi-literal that constructs a new value. Correspondingly, the postconditions of the five recursive calls to convert are simply the postcondition of convert, with the formal argument name replaced by the actual argument name, and the sixth postcondition describes the free atoms in the constructed value—these follow from the types and binding specifications used in the constructor. Writing these postconditions manually is a very mechanical task—the kind of task one would expect computers to do, and we will now show that that is indeed possible.

The proof obligations for function calls and quasi-literals can easily be inferred. We already saw in section 2.6 that proof obligations are generated only when we encounter either a function call or a quasi-literal. We also saw how the postcondition in a let-expression is used to transfer knowledge from the value expression to the body. Suppose the proof system is checking a function and encounters a `let`-expression:

$$\Gamma \vdash_{proof} \{H\} \left(\textbf{let}\ x\ \textbf{be}\ e_{val}\ \textbf{where}\ C\ \textbf{in}\ e_{body}\right) \{P\}$$

Now assume that e_{val} is either a quasi-literal or a function call. We look at the two cases individually:

- If $e_{val} = e^{qlit}$, then the proof obligation generated for the value expression will be: $\Gamma, \cdot : \tau; H \wedge \left(\cdot =_{\text{val}} e^{qlit}\right) \vDash_{\text{hyp}} C$, where τ is the type of e^{qlit}.
- If $e_{val} = (\textbf{call}\ f\ \overline{x_{actual}})$, then two proof obligations will be generated:

$$\Gamma; H \vDash_{\text{hyp}} \text{pre}(f)\overline{[x_{actual}/x_{formal}]}$$

$$\Gamma, \cdot : \tau; H \wedge \mathcal{F}(\cdot) \subseteq \mathcal{F}_e(\overline{x_{actual} : \tau_{formal}}) \wedge \text{post}(f)\overline{[x_{actual}/x_{formal}]} \vDash_{\text{hyp}} C$$

(Variables not explained here are to be seen in the context of the P-CALL rule).

The important proof obligations are those where C is the goal. Both have the following layout: $H \wedge A \vDash_{\text{hyp}} C$, where A is the additional knowledge generated from what we know about the expression. Now the P-LET rule branches to the body expression as follows:

$$\Gamma, x : \tau_{val} \vdash_{\text{proof}} \{H \wedge C[x/\cdot] \wedge \mathcal{F}(x) \subseteq \mathcal{F}_e(\Gamma)\}\ e_{body}\ \{P\}$$

Here, C represents the additional knowledge that we got from the value expression. We know what this additional knowledge actually is: A. If we set $C = A$, we transport all the knowledge we can get.

For e_{body}, at least one proof obligation will be generated. Consider an arbitrary proof obligation from those generated for e_{body}. It will look as follows:

$$\Gamma'; H \land C[x/\cdot] \land \mathcal{F}(x) \subseteq \mathcal{F}_e(\Gamma) \land H' \vDash_{\text{hyp}} P'$$

Γ' is the new environment. It is a superset of the environment the proof system had when it went on to traverse e_{body}, and a superset of the environment of the proof obligation for the value expression (modulo $[x/\cdot]$-renaming). This means that all names in the constraints refer to the same variables, hence we can ignore the environments from here on. H' is the part of the hypothesis that was added by the proof system while traversing e_{body}. P' is the postcondition of the proof obligation, which may or may not contain P (the proof obligation could come from a value expression of another `let`, or belong to a function call and assert that the precondition of the called function is satisfied).

We simplify the notation a bit and say that the proof condition looks like:

$$C[x/\cdot] \land R \vDash_{\text{hyp}} P'$$

Where

$$R ::= H \land H' \land \mathcal{F}(x) \subseteq \mathcal{F}_e(\Gamma)$$

We claim that there is no collection of clauses C' that can be used as a post-condition for our let, such that C' both follows from $H \wedge A$ and when used in the proof of the body a postcondition can be proven that cannot be proven when we use A—in short, C' cannot carry more additional information than A. In a more formal notation, that is:

(1) $H \wedge A \vDash_{\text{hyp}} C'$

(2) $C'[x/\cdot] \wedge R \vDash_{\text{hyp}} P'$

(3) $A[x/\cdot] \wedge R \nvDash_{\text{hyp}} P'$

Proof (by contradiction)

Because of (3), there must be a model $\mathcal{I}$, such that:

(4) $\mathcal{I} \vDash_{\text{hyp}} A[x/\cdot] \wedge R$

(5) $\mathcal{I} \nvDash_{\text{hyp}} P'$

From (4) and the definition of R, it follows that:

(6) $\mathcal{I} \vDash_{\text{hyp}} A[x/\cdot]$

(7) $\mathcal{I} \vDash_{\text{hyp}} R$

(8) $\mathcal{I} \vDash_{\text{hyp}} H$

From (5), it follows that:

(9) $\mathcal{I} \nvDash_{\text{hyp}} C'[x/\cdot] \wedge R$

From (7) and (8), it follows that:

(10) $\mathcal{I} \nvDash_{\text{hyp}} C'[x/\cdot]$

By construction, neither A nor C' nor H contains x, and neither H nor $\mathcal{I}$ contains $\cdot$, hence:

(11) $\mathcal{I}[\cdot/x] \vDash_{\text{hyp}} A$

(12) $\mathcal{I}[\cdot/x] \vDash_{\text{hyp}} H$

(13) $\mathcal{I}[\cdot/x] \nvDash_{\text{hyp}} C'$

This makes $\mathcal{I}[\cdot/x]$ a counterexample for $H \wedge A \vDash_{\text{hyp}} C'$, leading to a contradiction. ■

What this result shows us that the only constraint in the program in Example 3.2 that needs to be supplied by the programmer is the postcondition of the function. All other constraints can easily be inferred. On top of that, as we will see in section 4.4.1, this also enables a great deal of optimization: since $H \wedge A \vDash_{\text{hyp}} A$ is a tautology, we do not even need to check that proof obligation.

3.3 Errors can be explained in detail

If the proof system deems a program ok, we have some guarantees about our program: most importantly that it preserves α-equivalence, and also respects the other constraints we may have written down. On the other hand, if the proof system does not admit a program, we do not have any guarantees, and we usually want to fix this. So we ask a simple but very fundamental question:

Why?

Ultimately, the usability of any system like Romeo depends on the burden it places on the programmer to give it the information it needs, and the quality of its explanations when it fails to prove something. In this section, we explore what happens when proof obligations cannot be discharged, and what information we can provide to the programmer in such cases.

3.3.1 Postconditions that are not valid

In the way we have presented Romeo so far, the only possible cause for the proof system to not admit a program is that some postcondition cannot be proven valid.

As we discussed in section 3.1, we check postconditions clause by clause. That is, we have a hypothesis $H = H_1 \land H_2 \land \ldots \land H_n$ and a single goal clause $G = S_1 \; rel \; S_2$, where S_1 and S_2 are set formulas, and $rel \in \{=, \neq, \subseteq, \#\}$. We assert $H \land \neg G$ to establish the validity of $H \rightarrow G$. If $H \land \neg G$ has a model, we have a counterexample, and know a postcondition that is not valid. Although that postcondition may not come from an annotation that the programmer made, we always have a location in the code we can point to. We also can analyze the counterexample to tell the programmer what went wrong in more detail—this is what we will discuss in the next section. First, however, we will briefly discuss why we can always give a good source location and simple explanation for a failing postcondition.

If we are given a failing postcondition and the knowledge of where the postcondition and the proof obligation “came from”[1], we can always give the programmer precise source location information of what has failed.

In the simplest case, the postcondition was written as an annotation by the programmer—then we can simply point to that location and the expression that generated the proof obligation and say that the expression does not guarantee that the part of the postcondition that we point to holds.

If the postcondition was generated by the proof system, there are three possibilities why it could have been generated:

- There was a `fresh`-expression
- There was an `open`-expression
- There was a function call and we need to check its precondition.

In the first two cases, the generated proof obligation is about atoms that may not escape. If the proof obligation of a fresh-expression fails, this means that the atom introduced by it appears free in the result. We can point to the fresh-expression and again the sub-expression that generated the proof obligation and inform the programmer of this fact. Similarly, for an

[1] What do we mean by “came from”? Proof obligations are only generated at the leaves of the abstract syntax tree of a Romeo program. Therefore, they always corresponds to a particular function call or quasi-literal that produces a result that should satisfy the goals of the proof obligation.
For postconditions, they are either directly annotated by the programmer in either a function definition or a let-clause, or are generated by either the proof system or by the postcondition inference method described in section 3.2.

open-expression, at least one of the exposable atoms of the opened term is escaping (although we do not know which one)—apart from that, the case is the same as for fresh. In the case of a function call, the postcondition is actually related to the precondition of the called function, which was definitely written by the programmer.

Last, if the postcondition was generated by the inference method we presented in section 3.2, we know that it is trivially valid, so it will never fail, and we never have to give an explanation.

3.3.2 Analyzing Counterexamples

In CAL, counterexamples are finite. That means that when we get a model, we get a finite-size interpretation of the form $set_i = \{a_1 \dots a_n\}$ (the as are disjoint instances of the sort atom) for each set in the proof obligation that we were checking. As the model is a model for $H \wedge \neg G$, it is a model for $\neg G$, which is what we are really interested in. Recall that G is by construction limited to relate two set formulas S_1 and S_2, each of which contains set names and the operators $\cap$, $\cup$, and $\setminus$. Now that we know which atoms are in which set, we can actually evaluate S_1 and S_2. This makes it possible to point to individual problematic sets in all forms of G except if G represents an inequality constraint. If an inequality-constraint fails, all we know is that the two sides may be equal, but finding out which of the sets causes the equality contrary to the intentions of the programmer seems infeasible. However, as such a constraint is never generated, telling the programmer that the inequality they annotated may not hold should be sufficient.

For all other relations, we can find the culprits. First, we need to find the problematic atoms. Let $\mathcal{A}_1, \mathcal{A}_2$ be the sets of atoms in the results of evaluating S_1, S_2, respectively. Depending on the relation $rel \in \{=, \subseteq, \#\}$, the set of problematic atoms is defined as:

$$\mathcal{A}_p = \begin{cases} (\mathcal{A}_1 \cup \mathcal{A}_2) \setminus (\mathcal{A}_1 \cap \mathcal{A}_2), & G = S_1 = S_2 \\ \mathcal{A}_1 \setminus \mathcal{A}_2, & G = S_1 \subseteq S_2 \\ \mathcal{A}_1 \cap \mathcal{A}_2, & G = S_1 \ \# \ S_2 \end{cases}$$

Figure 3.3 – Finding problematic atoms

Each atom in $\mathcal{A}_p$ makes the proof obligation un-dischargeable on its own, which means we can use any atom in $\mathcal{A}_p$ to generate an explanation.

In any case, we need to find the possible sources of the atom a we choose. A simple way to do this is to take the set of source-sets $\mathcal{S} = \{set_i | a \in set_i\}$. Each set_i corresponds to some $\mathcal{F}_b(z)$ or $\mathcal{F}_r(z)$ in the original postcondition, meaning that our atom is a free binder or free reference, respectively, in a variable x or the result $(\cdot)$. We can now point the programmer to the failing proof obligation and the expression that does not satisfy it, and tell them that the reason for the failure is that there may be an atom that is a free reference in some zs and a free binder in some other zs. By looking at the proof obligation, and those variables, the programmer should be able to find out which variables are either missing the free atom or expose it although they should not.

$$\text{sources}(a, S)$$

$$= \begin{cases} \{s\}, & S = s \land a \in s \\ \text{sources}(a, S_1) \cup \text{sources}(a, S_2), & S = S_1 \cup S_2 \\ \text{sources}(a, S_1) \cup \text{sources}(a, S_2), & S = S_1 \cap S_2 \land \text{sources}(a, S_1) \neq \emptyset \land \text{sources}(a, S_2) \neq \\ \text{sources}(a, S_1), & S = S_1 \setminus S_2 \land \text{sources}(a, S_2) = \emptyset \\ \emptyset, & \text{otherwise} \end{cases}$$

Figure 3.4 – Narrowing down the sources of a problematic atom

3.3.3 More precision for sources of problematic atoms

A part of what the programmer has to do with the explanation they get as described in the previous section is re-trace the proof obligation to see how the problematic atom turns up in the result of each side of the goal. This can actually be done automatically, too.

Figure 3.4 shows a simple algorithm that takes an atom a and a set-formula S and returns those sets that are actually responsible for why a is in the result of S. The two important rules are those for intersection and set-difference. If we are at an intersection, both sides must contain sources of a, in which case the union of the sources on both sides is taken (not the intersection, which might be empty). If one of the sides does not contain a source of a, then the current sub-formula is not a source of a in the final result. The reasoning for set-difference is similar.

3.3.4 Unsatisfiable Postconditions

Getting a counterexample for a postcondition means that the postcondition cannot always be satisfied. But there actually may be cases where a postcondition can never be satisfied, for example because it contradicts one of the hypotheses. In such a case, giving the programmer a counterexample is a waste of their time because they need to figure out which atoms should appear where. They may try to fix the problem that causes the specific counterexample, which may result in just another counterexample being generated. We can avoid this scenario with a simple trick: we can ask the SMT solver whether the postcondition is satisfiable at all.

So far, we asserted $H \land \neg G$ and got a model which is our counterexample. We can assert $H \land G$ to see if that is satisfiable. If it is not, we can extract an unsatisfiable core of clauses in the pre- and postcondition, and show the programmer that those clauses will never be satisfiable together. Having to consider preconditions adds some complexity because they have more possible sources, but the reasoning is the same as the one we discussed for postconditions.

3.4 Summary

In this chapter, we have presented three key points that contribute to the usability of Romeo and thus the applicability of its ideas. First, we showed that the approximated proof obligations that Romeo produces are already in a format suitable for a direct translation to SMT solver input in a decidable form.

Second, we showed that many assertions can actually be inferred. This will make way to partially lift the restriction that only variables are accepted in many places where one would usually expect to see expressions. We will show how this works in the next chapter.

Third, we have discussed how errors in discharging the proof obligations generated by the proof system can be linked back to the source code that a programmer wrote. This is important because a system that just tells the user that something failed is not very useful.

We will discuss some caveats and experiences in the following chapters.

4 Implementation

Given Stansifer's core implementation of Romeo (containing everything discussed in chapter 2), we implemented a few extensions based on our observations in the previous chapter. The core Romeo implementation was written largely using PLT Redex [25], mixed with Racket code. Our implementation uses the same languages, as well as the SMT solver Z3 [26]. In this chapter, we introduce Romeo-L, a language implemented as a Racket #lang with ML-like syntax (close to Pottier's prototype implementation of pure FreshML [18]) that is compiled down to core Romeo. Its main advantage in contrast to core Romeo is its enhanced usability because of named types[1], use of constructors and inference of constraints as discussed in section 3.2—which also enables function calls and quasi-literals to be arbitrarily nested. This makes it much easier to test Romeo with examples and compare it to other systems. Such tests and comparisons will be discussed in chapter 5. We also explain our implementation of meaningful error messages based on the observations in section 3.3.

4.1 Romeo-L

We present Romeo-L, a "sugared" version of Romeo. Its main features are named types with constructors and what we will call *argument-expressions*. These are based on the observations in section 3.2 that many of assertions are inferable, namely those of quasi-literals and function calls. Argument-expressions represent either quasi-literals or function calls, and

[1] However, the names only serve as abbreviations. Type-checking is still done using the typing rules of Romeo, which are based on structural equality.

$$
\begin{array}{rcl}
Program & ::= & \overline{df}\ e \\
df \in Definitions & ::= & tD \mid fD \\
tD \in TypeDefs & ::= & \text{type } id_t \text{ is } \overline{\mid id_c\ \overline{id_t} \downarrow \beta \uparrow \beta} \\
fD \in FunDefs & ::= & \text{fun } id_f \left(\overline{id_a : id_t} \right) \text{ where } C \text{ returns } id_a : id_t \text{ where } C \text{ is } e \text{ end.} \\
e \in Expressions & ::= & e_a \mid \text{case } e_a \text{ of } \overline{\mid e_c} \text{ end.} \mid \text{fresh } \overline{id_a} \text{ in } e \mid \text{let } \overline{id_a = e \text{ where } C} \text{ in } e \mid \\
 & & \text{let } \overline{id_a = e_a} \text{ in } e \mid \\
 & & \text{if } e_a = e_a \text{ then } e \text{ else } e \\
e_a \in Arg\text{-}Exprs & ::= & id_a \mid id_c(\overline{e_a}) \mid id_f(\overline{e_a}) \\
e_c \in Case\text{-}Exprs & ::= & id_c\ id_a \Rightarrow e \mid \text{default} \Rightarrow e \\
C \in Constraints & ::= & \text{true} \mid s = s \mid s \neq s \mid s \subseteq s \mid s \# s \mid C \wedge C \\
s \in Sets & ::= & \emptyset \mid \mathcal{F}(id_a) \mid \mathcal{F}_x(id_a) \mid \mathcal{F}_r(id_a) \mid \mathcal{F}_b(id_a) \mid (s) \mid s \cup s \mid s \cap s \mid s \setminus s \\
\beta & ::= & \overline{natural}^{1} \\
id_{\_} \in Identifiers^{2} & &
\end{array}
$$

Figure 4.1 – The syntax of Romeo-L

they can appear wherever previously a variable was required as a reference. All expressions where argument-expressions appear instead of the simple variables that are required in Romeo's syntax can be wrapped in `let`-expressions with automatically inferred postconditions.

4.1.1 Syntax

Figure 4.1 shows the syntax of Romeo-L. It is actually a bit richer than presented here, because every Unicode symbol has an ASCII equivalent (e.g. ∧ can be written as &&). We chose

[1] We use the same simplification here that we already used in the presentation of Romeo in chapter 2. In the actual implementation, Romeo-L offers the same operators as Romeo.
[2] The sets of identifiers need not be distinct. There is only one possible place where identifiers may be ambiguous: in argument-expressions, the notation for function calls and constructor applications are the same. In case of an ambiguity, this is interpreted as a function call.

the Unicode symbols for the presentation here to match the syntax of Romeo. The formulation of constraints and sets is almost identical to Romeo, expressions are written in a more ML-like syntax.

Type definitions are new and introduce named types. Every type is seen as a union of one or more products, each case has its own unique constructor. Import and export specifications are the same as in core Romeo. There are two pre-defined special types: `binder` and `reference`—they represent `Binder` and `Ref`, respectively.

Function definitions have a slightly different notation, but contain almost the same elements. The only addition is a name for the result of the function, which can be used in the postcondition. This replaces $\cdot$. If the result of the function is not needed in any constraint, the name can be omitted. Constraints can also be omitted and will default to $true$.

Expressions e are roughly the same as in core Romeo. Function calls $id_f(\overline{e_a})$ are part of the argument-expressions, and `case` and `open` are merged in the `case`-construct. A `case`-expression can contain up to one case per constructor of the type of its argument, plus a `default` case (which must be there if not all constructors are covered). Each case for a constructor must mention as many new variables that are bound to the fields of that case. The constructs for `fresh` and `let` are slightly generalized by letting them have multiple arguments that will be converted into several nested expressions. We also have an additional version of `let`. It allows the programmer to omit the postcondition, but accepts only argument-expressions for its value expression. Hence it can automatically infer the right postcondition. Last, `if`, like `case`, is extended to allow argument-expressions instead of just variables as arguments.

Finally, there are argument-expressions e_a. They represent quasi-literals and function calls. Note that there is no equivalent to the ($\textbf{ref}\,x$) quasi-literal. This is because it can automatically be placed at the right positions. Recall that Romeo restricted many positions where one would expect expressions to only accept variables, which made it necessary to have lots of `let`-expressions. Argument-expressions enable us to keep the promise that we would partially lift this restriction. In Romeo-L, argument-expressions can be used in all these previously restricted positions.

4.1.2 Semantics

Romeo-L is in principle just syntactic sugar for Romeo. Therefore, the semantics of Romeo-L is given by a combination of the translation from Romeo-L to Romeo and the semantics of Romeo.

4.1.3 Translation

The biggest complication in a translation from Romeo-L to core Romeo is the strategy of placing let-expressions. We will start with three restrictions on Romeo-L which we will lift gradually. First, all places in the syntax that were restricted to variables in Romeo are restricted to variables again in Romeo-L. Roughly, that means that argument expressions cannot be nested. The only places where they may now appear are in places where normal expressions are accepted, and in the value expression of a `let` without annotated postcondition. Second, there is no difference in terms of types between `Binders` and `Refs`. Both references and binders are converted to type `Atom`, which may be either a binder or a reference (this was true in an earlier version of Romeo). As we will see, in order to be able to lift this restriction, we first need to lift the first one. Third: `fresh` and `let` can each introduce only one name.

Simple Translation without Nesting

The translation works on individual function definitions, which are the main components of a Romeo program (and the only one that we are interested in). The function to convert function definitions just has to call the translation functions for all its components and piece them back together. Constraints do not have to be translated—they already come in the right format from the parser (all the parser has to do is replace the name of the return variable with $\cdot$). Types are straightforward to convert, the most complicated thing that that translation function has to do is to keep track of already encountered recursive types to use the right type variables in the right positions.

Expressions are translated by two main functions: one for the "real" expressions e, and one for argument expressions e_a. Both functions have access to the function and type definitions. Apart from that, the former simply takes an expression and returns a translated expression, whereas the latter takes an argument expression, a variable name to bind it to, and an expression that possibly contains a reference to the aforementioned variable name. From these three components (or rather, their translations), it will generate a `let`-expression.

Figure 4.2 shows this basic translation. Most translations are straightforward. Case-expressions and constructors are expanded into slightly larger expressions which we omitted in the

$$
\begin{array}{rcll}
[\![x]\!]^e & = & x & \\
[\![e_a]\!]^e & = & [\![e_a]\!]^a_x(x) & x \text{ fresh} \\
[\![\text{let } x = e_1 \text{ where } C \text{ in } e_2]\!]^e & = & (\textbf{let } x \textbf{ be } [\![e_1]\!]^e \textbf{ where } C \textbf{ in } [\![e_2]\!]^e) & \\
[\![\text{let } x = e_a \text{ in } e]\!]^e & = & [\![e_a]\!]^a_x(e) & \\
\left[\!\left[\text{case } x \text{ of } \overline{\mid c\ \overline{x_\iota} \Rightarrow e} \text{ end.}\right]\!\right]^e & = & [\textbf{case} + \textbf{open}] & \text{(see below)} \\
[\![\text{fresh } x \text{ in } e]\!]^e & = & (\textbf{fresh } x \textbf{ in } [\![e]\!]^e) & \\
[\![if\ x_1 = x_2\ then\ e_1\ else\ e_2]\!]^e & = & (\textbf{if } x_1 \textbf{ equals } x_2 \textbf{ then } [\![e_1]\!]^e \textbf{ else } [\![e_2]\!]^e) & \\
 & & & \\
[\![x]\!]^a_{x_b}(e) & = & (\textbf{let } x_b \textbf{ be } x \textbf{ where } C \textbf{ in } [\![e]\!]^e) & C \text{ inferred} \\
[\![f(\bar{x})]\!]^a_{x_b}(e) & = & (\textbf{let } x_b \textbf{ be } (\textbf{call } f\ \bar{x}) \textbf{ where } C \textbf{ in } [\![e]\!]^e) & C \text{ inferred} \\
[\![c(\bar{x})]\!]^a_{x_b}(e) & = & \textbf{let } \ldots\ [\textbf{inj0/inj1} + \textbf{prod}] & \text{(see below)}
\end{array}
$$

Figure 4.2 – Basic translation of Romeo-L to Romeo

figure for space and clarity, but we will give an intuition below. The most interesting lines are the conversion of a `let` without annotated postcondition and the one with an arbitrary argument expression that is not a variable. What we can see is that the argument expression translation function exactly models that interesting `let`. While that `let` produces a Romeo-`let` that was wanted by the user, an argument-expression in a simple expression-position produces a superfluous `let` of the form (**let** x_b **...** **in** x_b), which could just be replaced by its value expression. However, this does not change the semantics of the program, so we will stick with it for simplicity.

Cs are inferred almost[1] exactly as described in section 3.2. A constructor is desugared to some number of injections plus one **prod** at the end—that **prod** contains the necessary information to generate the constraint here. For function calls, we retrieve the postcondition of function f and replace the formal parameter names with the actual parameter names, and for some variable we only need to assert that the "new" variable's binders and references are equal to the binders and references of the "old" variable.

Basic Nesting

Next, we will allow argument-expressions to appear in `case`-expressions instead of just variables. This is a rather easy change: all we need to do is to generate a new variable name, replace the argument-expression with that variable name and wrap the `case`-expression in a `let`-statement that declares the generated variables to be the result of the argument-expression.

Thus, the translation goes as follows:

$$\left[\!\left[\text{case } e_a \text{ of } \overline{\mid c\ \overline{x_\iota} \Rightarrow e} \text{ end.}\right]\!\right]^e = [\![e_a]\!]^a_{x_{new}}\left(\text{case } x_{new} \text{ of } \overline{\mid c\ \overline{x_\iota} \Rightarrow e} \text{ end.}\right)$$

Recall that in Romeo(-L), an `if` statement has four parts because it always has to compare two atoms. Having argument-expressions instead of the two variables in an `if`-expression seems to be easy, too, and in principle it is—we only need two `let`-expressions. However, this

[1] The only difference is that the $=_{val}$-constraints that would be generated for variables are instead generated in already expanded form. That is, whereas $z =_{val} e^{qlit}$ would cause every $\mathcal{F}_b(z)$ to be replaced with some set-expression s_b (similarly for the subscript r), we generate a constraint $\mathcal{F}_b(z) = s_b$, which is equivalent, and matches the syntax of Romeo (since $=_{val}$ is designated for internal use only).

$$
\begin{array}{rcl}
[\![e_a]\!]^a_{x_b}(e) & = & \text{t-argexp}([x, e_a]\colon \emptyset, e) \\
\text{t-argexp}(\emptyset, e) & = & [\![e]\!]^e \\
\text{t-argexp}([x_b, x]\colon r, e) & = & \big(\textbf{let } x_b \textbf{ be } x \textbf{ where } C \textbf{ in } \text{t-argexp}(r, e)\big) \\
\text{t-argexp}([x_b, f(\bar{x})]\colon r, e) & = & (\textbf{let } x_b \textbf{ be } (\textbf{call } f\ \bar{x}) \textbf{ where } C \textbf{ in } \text{t-argexp}(r, e)) \\
\text{t-argexp}([x_b, f(\overline{e_a})]\colon r, e) & = & \text{t-argexp}(\overline{[x_{new}, e_a]}\colon [x_b, f(\overline{x_{new}})]\colon r, e) \\
\text{t-argexp}([x_b, c(\bar{x})]\colon r, e) & = & \textbf{let } \dots\ [\textbf{inj0/inj1} + \textbf{prod}] \\
\text{t-argexp}([x_b, c(\overline{e_a})]\colon r, e) & = & \text{t-argexp}(\overline{[x_{new}, e_a]}\colon [x_b, c(\overline{x_{new}})]\colon r, e)
\end{array}
$$

$$
\begin{array}{rcl}
[\![if\ e_{a_1} = e_{a_2}\ then\ e_1\ else\ e_2]\!]^e & = & \text{t-argexp}([x_{new_1}, e_{a_1}]\colon [x_{new_2}, e_{a_2}]\colon \emptyset, e) \\
 & & e = (if\ x_{new_1} = x_{new_2}\ then\ e_1\ else\ e_2)
\end{array}
$$

Figure 4.3 – Extended argument-expression translation function, and new translation of `if`

is not as easily expressed using the translation function we have for now, so we defer it to the next step.

Arbitrary Nesting

Of our restrictions on the places where argument-expressions can appear (vis-à-vis the syntax of Romeo-L presented in Figure 4.1), those for `if` and those for argument-expressions themselves are left. `if` needs up to two, argument-expressions any number ≥ 0 of arguments—and they may even be nested.

Both needs are addressed with a simple extension of the argument-expression translation function, shown in Figure 4.3. Instead of just one-argument-expression, it takes a stack of variable-argument-expression-pairs as input. We use the overbar notation for stacks similar to lists, and colon to represent an equivalent to concatenation. In each step, the function looks at the stack. If it is empty, all argument-expressions have been evaluated and we can proceed by translating the wrapped expression. Else, we look at argument-expression in the top element. If it is a variable, we do the same thing as in the old translation function. Similarly, if we have a function call or constructor application whose arguments are all variables, we do the same thing as before.

If we have a function call or constructor application that contains some argument-expression that is not a variable, we generate a new set of variables, one for each such argument-expression. We put the function call/constructor application back on the stack, with the argument-expressions replaced by the generated variables, and then put pairings of the argument-expressions with those variables on top of that. This stacking strategy ensures that all sub-expressions are evaluated before they are needed, generating a deeply nested tree of `let`-expressions. However, nesting only happens in the bodies of those `let`-expressions. The value-part is always just one quasi-literal or function call, for which we can easily infer the postcondition.

This whole scheme is of course optimizable in a straightforward way by not pushing every argument-expression on the stack, but instead only those that are not variables already.

Binders and References

The first restriction (only variables in many positions) has fallen. This means it is time to reintroduce the distinction between binders and references. The core problem here is that Romeo explicitly distinguishes the two and their types. The type-checker does not allow a binder and a reference to be compared, but they are both atoms, and it must be possible to compare atoms. A binder can be turned into a reference via the `ref` quasi-literal. But `ref` is not included in Romeo-L's syntax, because it can actually be inferred.

All we have to do is a bit of type inference. We need to keep a type-environment to be able to determine what type an expression actually has (due to our translation, only variables are in returning positions), and we need to keep around an expectation of what type an expression should have (although this expectation may also be `any`, for the value-branches of `let`-expressions, where we will never know what to expect if the bound variable is not used). Then, we just care about one special case: if the returned variable x has type Binder, but we actually expect type Reference, then we replace x with

$$(\textbf{let}\ x_{new}\ \textbf{be}\ (ref\ x)\ \textbf{where}\ x_{new} =_{\text{val}} (ref\ x)\ \textbf{in}\ x_{new})$$

(imagine the $=_{\text{val}}$-constraint properly expanded). All other combinations of non-matching types are still illegal. For an `if`-expression, we can expect both sides of the equality to be references, and the inference mechanism will ensure that.

Without Restrictions

The last restriction left is that `fresh` and `let` can each introduce only one name. This was mainly to avoid unnecessary complications in the presentation of the translation. It is easily introduced by simply nesting the multiple instances. `let` therefore actually has let*-like semantics.

Case Expressions and Constructors

Finally, we give an intuition of translating case-expressions and constructors. For case expressions, it is important to generate the right number of case-open-combinations. Recall that a type consists of a non-empty list of constructor definitions. We expect the cases of the case-expression to come in the same order and amount (using the constructor names, they can be properly ordered beforehand, and default can be used to insert missing cases). Because the types are translated in a right-associative way, we can (and have to) translate the case-expressions in a right-associative way, too. Each first sub-expression of a binary Romeo case-expression will be an open-expression, the second part is generated by recurring on the rest of the cases.

For constructors, we have to wrap a product constructor into the right number of injections. We do this by looking for the current constructor in the list of constructors of the appropriate type. While we have not found it, we place inj_1s, where τ_0 is the type of the constructor we are currently looking at. When we encounter the constructor we were looking for, we place an inj_0, where τ_1 is the type of the rest of the constructors, or no injection at all if the constructor is the last one in its type.

4.2 Interaction with the SMT Solver

As laid out in section 3.1, the proof obligations (after approximation) are already in a form that is simple to convert to SMT solver input. We use the SMT solver Z3 in an interactive mode, which enables us to react to errors and extract information about them. Figure 4.4 shows the general scheme of this interaction. A push operation creates a new sub-scope of

```
for each proof obligation Γ;H ⊨hyp P :
  push. ;; new scope
  for each clause hc in H :
    assert(hc).
  for each clause pc in P :
    push. ;; new scope
    assert(not pc).
    s ≔ check-sat.
    if s = sat then
      m ≔ get-model.
      pop.        ;; this retracts assert(not pc)
      push.       ;; new scope
      assert(pc).
      s’ ≔ check-sat.
      if s’ = sat then
        process-counterexample(m).
      else
        u ≔ get-unsat-core.
        process-unsat-core(u).
    pop.          ;; retracts either assert(not pc) or assert(pc)
  pop.            ;; retracts all clauses in the hypothesis
```

Figure 4.4 – General scheme of interaction with the SMT solver

assertions that are retracted by the corresponding pop operation. This enables us to use the same set of set-names for each proof obligation without having to worry about them interfering. For each proof obligation, we first assert all clauses of the hypothesis. Then, we check each goal clause individually, again using push/pop to be able to use the same set of hypothesis assertions for each goal clause.

If the negated goal clause in combination with the hypothesis is unsatisfiable, all is well. If, on the other hand, it is satisfiable, we can extract a model as a counterexample. As discussed in section 3.3.4, it may well be that the goal itself is unsatisfiable in conjunction with the hypothesis, so we get rid of the assertion of its negation and create a new scope with pop/push, and assert the positive version of the goal clause. If the positive version of the goal clause is satisfiable, we really only have a counterexample that we need to explain to the user. Otherwise, we can extract the unsatisfiable core and tell the programmer which elements of his code will never work together.

Both `get-model` and `get-unsat-core` are operations provided directly by Z3.

4.3 Explaining Errors

The most important ideas on how to explain errors were already covered in section 3.3. In this section, we discuss a few implementation-specific details.

4.3.1 Bypassing Romeo

We have been using Stansifer's implementation of Romeo as a trusted base so far—for example, we did not change the proof system to infer postconditions and generate the right proof obligations. We instead generated standard Romeo code. That works well when the programs are correct, but when we want to explain errors, we need some more information at the point where we encounter the error, that is, in the SMT interface. In particular, we want to know where each proof obligation and its clauses come from. The proof obligations that Romeo generates do not contain that information.

To protect the integrity of our trusted base, we opted to leave it unchanged. Instead, we are going to use our knowledge of its inner workings. This means that we almost copy everything Romeo does, and generate the information we need in the same shape as Romeo generates the proof obligations. We then pass it to the SMT interface along with the proof obligations Romeo generated, where we now can match up the pieces that Romeo generated with those that we generated in Romeo-L. To be precise, this means that we predict every single goal clause that will be generated, and generate the corresponding information for each goal clause, such that we know where in the source code it came from. This can be integrated in the translation process described in section 4.1.3. Of course, this strategy is highly fragile—any non-trivial change in how Romeo generates proof obligations will likely require a corresponding change in the code that generates the additional information, lest matching of the two parts will fail. On the other hand, one can see this as a way of preparing to make the implementation of Romeo-L independent from the implementation of Romeo: given the work the translation has to do, it only requires some minor changes for it to be able to actually generate the proof obligations on its own.

Currently, we predict the number, sequence and shape (what goal clauses) of the proof obligations, and collect the necessary information about source locations and involved variables. The matching function in the SMT interface does some basic checks to ensure that the predictions match the proof obligations that were generated (e.g. checking that the predicted number of goal clauses matches the actual number of goal clauses).

4.3.2 Handling Generated Variables

The important parts of analyzing counterexamples were explained in section 3.3.2. There is only one complication: many of the variables may be generated. Implemented naïvely, the programmer would get an error message that there may be an atom that is a free reference in the variable `x474392`, which does not appear in the code. We can keep track of what the variable was generated for and say that it is the result of the function call xyz. But we can do slightly better: we can track which variables flow into the computation. To be precise, we have to track where the binders and references come from. In general, we can determine this exactly for variables (both if they are just used as a reference and if they are used as temporary variables in translations of `case`-expressions) and very precisely for constructors.

For function calls, it is not that simple—we would have to use the postcondition to reason about where binders and references come from. But the postcondition might consist of an arbitrary number of clauses, including disjointness, inequality and subset-constraints. Or it may just be `true`. The only thing we know for sure is that the free atoms in the result must be a subset of the union of free atoms in the arguments. Hence, we approximate this by saying that all the variables used as arguments to the function are potential sources of both free binders and free references. Similarly to Wand [27], we point to a small set of variables, at least one of which is responsible for the error.

To get back to our example—if our problematic atom may be a free reference in `x474392`, we go and recursively collect all the sources of free references for it. This search is guaranteed to terminate, because the generated variables form a tree of dependencies, not a graph, and at some point we get to a variable that actually appears in the code, which has no more dependencies.

```
Abs(x,App(y,z))
```

Example 4.1 – A nested constructor

Consider Example 4.1. It constructs the term $\lambda x.\,(y\ z)$ (`Abs` being the constructor for λ-abstractions, and `App` for applications). In the translation to Romeo, this expression is wrapped into at least one `let`-expression that binds the result of the application constructor. Assume that the whole expression is wrapped in a `let`, and is given the name `x474392`, and that the name of the result of the application constructor is `app123`. From the type of the λ-abstraction, we know that its left component (the `x`) is a Binder, so there are no references in there. But its right component may have free references. In the construction of this particular λ-abstraction instance, the variable on the right side was `app123`. That means we are now looking for the sources of free references of `app123`. An application can get its free references from both components, which were `y` and `z`. In our scenario, those have type Ref and thus have no further dependencies, so the possible sources of our problematic atom are `y` and `z`.

One last caveat is again the `ref` quasi-literal. If an atom is converted from a binder to a reference, the references in the new variable are dependent on the binders in the old one. We have to remember that this switch happened to be able to search at the right places.

4.4 Experimental Extensions

The way that we are bypassing core Romeo to transport information from Romeo-L to the SMT interface enables us to create a variety of extensions, which we will discuss in this section.

4.4.1 Not Checking Inferred Goals

As we saw in section 3.2, inferred constraints are trivially valid, since they are part of the precondition. We can therefore tell the SMT interface not to check the proof obligations that result from value-expressions of `let`-expressions where the postcondition was inferred, thereby decreasing the run-time of the whole checking process.

4.4.2 Handling Absurdity

Some situations in the code may be absurd, that is, they can never be reached because the preconditions are not satisfiable, which means that the execution system will never get to this point—it would fault before it started to evaluate the leaf of an absurd expression. Example 4.2 shows how an absurd case could look like. When we get to the inner `if`, we already established that $x = y$, hence the `else`-part of the inner `if` is absurd.

```
if x = y then if x = y then ___ else _!_ else ___
```

Example 4.2 – Absurdity: the inner else-part is absurd and can never be reached

How these situations should be handled is not entirely clear and depends on the situation. We decided to warn programmers about absurd preconditions. In order to do that, we add a check for satisfiability after the loop that asserts all the clauses of the hypothesis in our algorithm presented in Figure 4.4. If the precondition is unsatisfiable, we print a warning. Note that this is affected by the optimization presented in the previous section: if we filter out all proof obligations that were inferred, they may not be checked for absurdity.

4.4.3 Absurd and Fail

Among other things, pure FreshML [18] contains the two statements `absurd` and `fail`. `absurd` could be written in the absurd position in Example 4.2, and would ask the SMT solver to actually prove that the preconditions there are unsatisfiable. `fail` on the other hand is a dynamic construct. The execution system will stop when it reaches a `fail` statement. In exchange, a `fail` statement terminates a branch of an expression without generating any proof obligations—this is useful for default branches of case-expressions or other non-absurd, but problematic scenarios. For example, when we write an interpreter for the λ-calculus, at some point we only want abstractions on the left-hand side of an application. We can handle the case of an abstraction, and have a default case for all other possible terms, where we let the program fail. That way, we do not have to come up with some expressions for these error cases that still satisfy the postconditions of the function.

At present, Romeo does not support `absurd` or `fail`, but we can emulate them, which enables us to translate a few more examples from the pure FreshML homepage (see chapter 5). In order to do that, we need to do two things: we need to transform them into code that the type checker of Romeo admits, and we need to make sure that the SMT interface knows which proof obligations should have absurd preconditions and which proof obligations to ignore altogether.

We can use the way we already transport extra data to the SMT interface (see section 4.3.1) to have the SMT solver do the right thing for both `absurd` and `fail`. To be able to generate code that the type checker admits, we have to trick the type checker. The programmer needs to annotate `fail` and `absurd` with a type, and the translation process will then replace them with a call to a function that takes no parameters and has the return type that the programmer annotated. We never check these functions, but we put them into the dictionary of functions that we give the type checker for reference so it thinks those functions actually exist. Programs generated this way are not really valid Romeo programs anymore—they cannot be executed—but they can be statically checked so we can see if they would work at least in theory.

5 Experimental Results

5.1 About our Experiments

We conducted some experiments to test the implementation of Romeo-L. Our thanks go to François Pottier, who published a number of examples along with pure FreshML on his homepage [28] that we could use to evaluate our own system. We also wrote a few smaller macros and some erroneous versions of them to test the error messages. For hard numbers, we were mostly interested in constraints. How many of them had to be annotated? How many could be generated? We also measured how long it takes to check our different examples, and how long the individual phases of the checking process take.

5.2 Translated pure FreshML Examples

We were able to translate 9 out of 14 examples on the FreshML web page (see Figure 5.1). In the cases that could be translated, the translation was mostly straightforward and was only about correcting syntactic differences. On the other hand, the reason for why the other pro-

Example	Romeo-L	Example	Romeo-L	Example	Romeo-L
anf-direct	YES	hoist	NO	metaml	YES
anf	YES	comb	YES	nbe	YES
atomlist	NO	cps	YES	nbe-delayed	YES
callcc	NO	debruijn	YES	typed-nbe	YES
cc	NO	interpreter	NO		

Figure 5.1 – Overview of examples from FreshML homepage and whether they could be easily translated

grams could not be translated was that Pottier's prototype implementation of FreshML introduces some more extensions that we could not easily reproduce, such as constraints for type cases (asserting inequality of free names between separate parts of a product) and return values that are paired with Boolean values which are used in an implication in the postconditions. Implementing those as experimental extensions (in fact, we implemented `absurd` and `fail` in order to be able to translate some of these examples) is surely possible. However, in contrast to the experimental extensions that we have actually implemented, these other extensions would require extensive surgery in the output of the proof system, thereby removing our theoretical foundation.

```
type expr/base is
  | Var/base reference
  | Abs/base binder expr/base ↓0
  | App/base expr/base expr/base
  | Set/base reference expr/base
  | Let/base binder expr/base expr/base ↓0
  | Seq/base expr/base expr/base
end.

type expr is
  | Var reference
  | Abs binder expr ↓0
  | App expr expr
  | Set reference expr
  | Let binder expr expr↓0
  | Seq expr expr
  | Swap reference reference
end.

fun expand (e : expr) returns r : expr/base where fr(r) = fr(e) is
  case e of
  | Var x => Var/base(x)
  | Abs x ex => Abs/base(x,expand(ex))
  | App e1 e2 => App/base(expand(e1),expand(e2))
  | Set x ex => Set/base(x,expand(ex))
  | Let x vex bex => Let/base(x,expand(vex),expand(bex))
  | Seq e1 e2 => Seq/base(expand(e1),expand(e2))
  | Swap x1 x2 => expand(swap(x1 x2))
  end.
end.

fun swap (x1 : reference, x2: reference) returns r : expr
    where fr(r) = fr(x1) ∪ fr(x2) is
      fresh tmp in
       Let(tmp,Var(x1),Seq(Set(x1,Var(x2)),Set(x2,Var(tmp))))
end.
```

Example 5.1 – Expressing the expansion of the swap-macro

5.3 Macros and Other Small Functions

Since the main goal of Romeo is to reason about hygiene in macros, we also wrote the standard macros from the literature about hygiene [4] [5] [6] [10]. Those macros are: or, swap, let* and letrec (transcribed from Herman's version [7]). We also tested for the aliasing problem (see section 2.5) and implemented β-reduction with capture-avoiding substitution. All of

```
type expr is
  | Var reference
  | Abs binder expr↓(0)
  | App expr expr
  | Letstar let-clauses expr↓(0)
end.

type let-clauses is
  | LCNone
  | LCBind binder expr let-clauses↓(0) ⇑(0 2)
end.

fun expand [see Example 5.1]

fun letstar(lc : let-clauses, e : expr) returns out : expr
 where fr(out) = fr(lc) ∪ (fr(e)\fb(lc)) is
  case lc of
  | LCNone => e
  | LCBind x be rest => App(Abs(x,Letstar(rest,e)),be)
  end.
end.
```

Example 5.2 – Let - a recursive macro*

these macros could be verified by Romeo. In section 5.5, we review a few error messages that came up while writing these macros.

The way we formulated the macros follows a general scheme: we specify one type that describes terms of the base language, and another type that describes terms of the extended language that contains the macro. Finally, there is a macro expansion function that takes a term of the extended language and converts it into a term in the base language by recursively mapping all terms in the extended language to their base language counterparts, if they exist, else it calls the respective macro handling functions.

Example 5.1 shows a complete example of expressing such a macro for the swap-macro. Terms of the base language are of type `expr/base`, terms of the extended language have type `expr`. The function expand takes an expression of the extended language and recursively turns it into an expression of the base language. The only really interesting case is the last one, where we encounter the `swap` macro. Here, we do the next expansion step after having expanded the macro. In the case of `swap`, that is not significant, but other macros we will see will emit code that contains another macro that has to be expanded. Finally, the `swap`-function models the `swap` macro's behavior. We generate a fresh variable for intermediate storage and then we do the swap. Generating a fresh variable ensures that there will be no name clashes with that macro.

Example 5.2 shows `let*`, which is both interesting for its binding structure that Romeo can easily model and for being implemented as a recursive macro. It actually generates a `Letstar`-term instead of directly calling itself—the `expand`-function will do that.

Finally, Example 5.3 shows beta-reduction and capture-avoiding substitution. This example is interesting because it shows the automated freshening that Romeo does, similarly to FreshML. Notice that in the abstraction case of capture-avoiding substitution, we do not need to check whether x and y are equal—the system will use the information from the type system to make sure (by proper renaming if need be) that the two have distinct names.

```
type term is
  | Var reference
  | Abs binder term↓(0)
  | App term term
end.

fun reduce( t: term) returns r : term is
  case t of
  | App t1 t2 =>
    case reduce(t1) of
    | Abs x tt => reduce(subst(tt,x,t2))
    | default => fail term
    end.
  | default => t
  end.
end.

fun subst( t: term, x : binder, arg : term) returns r : term where fr(r) ⊆
(fr(t) \ fb(x)) ∪ fr(arg) is
 case t of
 | Var y => if x = y then arg else t
 | Abs y tt => Abs(y,subst(tt,x,arg))
 | App t1 t2 => App(subst(t1,x,arg),subst(t2,x,arg))
 end.
end.
```

Example 5.3 - β-reduction and capture-avoiding substitution

5.4 Inferred Annotation Ratio and Running Time

Figure 5.2 shows data collected when running the translated pure FreshML examples and—below the thick black separator—the other programs we implemented. We show the total number of proof obligations generated for each example, but then split them up into goal clauses, that is, all the ∧-connected parts of the postconditions. We can see that on average, proof obligations have about two goal clauses. Of these goal clauses, we manually counted how many were annotated in the source code. Although we also wrote a small number of preconditions, we ignored them for this presentation. We automatically measured the numbers of proof obligations and goal clauses that were generated and the numbers of those that

Test Case	# Proof obligations	Goal Clauses						Run-Time				
		Total	Annotated	Inferred	Discharged	Savings (SMT)	Savings (An-	Total (sec)	Translation	Generation (%)	SimplAp-	SMT (%)
anf-direct	69	132	0	86	46	65%	100%	47.1	1.6%	14.0%	83.1%	1.3%
anf	74	132	0	76	56	58%	100%	95.9	3.0%	17.4%	79.1%	0.5%
comb	57	102	1	69	33	68%	99%	6.5	5.6%	19.2%	69.1%	6.1%
cps	74	145	0	106	39	73%	100%	30.8	2.3%	14.5%	80.1%	3.2%
debruijn	56	98	1	61	37	62%	98%	4.8	6.3%	15.6%	72.3%	5.8%
metaml	298	521	8	313	208	60%	98%	2044.9	0.4%	16.8%	82.7%	0.1%
nbe	53	97	1	56	41	58%	98%	12.9	3.9%	16.8%	76.7%	2.6%
nbe-delayed	59	108	1	62	46	57%	98%	39.6	5.2%	6.0%	87.8%	1.0%
typed-nbe	83	145	4	87	58	60%	96%	28.1	3.0%	8.9%	86.4%	1.7%
alias	17	31	1	20	11	65%	95%	2.1	9.2%	14.9%	69.1%	6.8%
β-reduction	26	43	1	24	19	56%	96%	2.8	11.5%	10.8%	74.1%	3.6%
letrec	98	178	7	117	61	66%	94%	183.6	1.2%	13.5%	85.0%	0.4%
let*	22	39	2	22	17	56%	92%	5.1	18.3%	9.6%	69.3%	2.7%
or	40	70	2	44	26	63%	96%	24.176	1.5%	17.1%	80.7%	0.7%
swap	42	74	2	48	26	65%	96%	15.421	1.9%	16.9%	78.9%	2.3%

Figure 5.2 – Distribution of sources of goal clauses and running times for pure FreshML examples and standard macros

were actually discharged. The ones that were not discharged must have been flagged as ignorable, which are exactly those constraints that were inferred. The Savings columns show what percentage of goals did not have to be discharged because they were flagged as ignorable (SMT), and what percentage of user annotations could be inferred (Annot.).

We also automatically measured the running times of the various phases of checking in our implementation. We used the built-in Racket `time-apply` construct to obtain these numbers, and used the "real" time component for this chart. The actual times vary somewhat when checking the same program more than once, but the distribution among the different phases is largely the same. We ran the checks on a machine that has an Intel Core i7 dual-core processor with 2.7 GHz per core and 8GB of main memory running Racket 5.3.3 (limited to 2GB of memory) on Windows 7.

Benefits of inferred constraints

What we see from the relations between total goal clauses and inferred goal clauses is that we can save about 60% of the work for the SMT solver by not discharging inferred goals. Similarly, the savings in proof obligations that would otherwise have to be annotated are huge.

There is one caveat to these numbers, though: code-generation could be made more efficient, pressing down the number of inferred and therefore total proof obligations. This would reduce the percentage of savings in annotated constraints. In particular, all argument-expressions that are not simply a variable are always nested into a let, even if they are in the return position. Also, multiple nested constructors could be translated into nested quasi-literals, which as of now is not done—each constructor causes at least one let-expression to be generated. We only coded a few of the examples manually in Romeo, but we estimate that more intelligent code generation could in reduce the number of inferred constraints by about 50%. The translation of nested constructors is the biggest source of inefficiency, whereas nested function calls only incur the overhead once—in the return position. However, the number of inferred constraints would still grow drastically with the size of functions, while the number of constraints that have to be annotated are roughly linear (with a very small constant multiplier) in the number of functions.

Running time

A major concern about using an SMT solver and an NP-complete implementation of set theory may be that running time could explode for larger programs. Based on our measurements, this is not the case—or at least, it is not the SMT solver's fault: we see that in the longest-running example, the Meta-ML interpreter, the portion of time spent interacting with the SMT solver is by far the lowest. The explanation for why NP-completeness does not play a huge role here is that because of the way our proof obligations are structured, we naturally use a technique called Verification Condition Splitting [29]. There, constraints that are given to an SMT solver are split along the control flow of the program to get smaller problems for the SMT solver to solve—which is similar to what we do. We even prove each goal clause separately, and the size of the input for such a goal is related to the complexity of the function it belongs to. As a design principle, functions should not be overly complex, and if they are not, the time spent interacting with the SMT solver with grow linearly with the number of functions.

So why does checking our small examples still take quite a while? The answer is that the implementation is inefficient, in large parts because of the heavy use of PLT Redex [25]. The code to simplify and approximate proof obligations and then translate them to SMT solver input takes by far the largest chunk of time, most probably because it does a lot of pattern matching against large data structures. The benefit of this is that the algorithms can be written a way that is very close to mathematical notation, which is perfect for our purposes. Of course, a more realistic implementation of this system could and should implement these parts in a more efficient way. It may even suffice rewrite this part of the program in standard Racket.

5.5 Error Messages

In this section, we will review some of the error messages our system generates and how they can be interpreted.

5.5.1 Free References

Example 5.4 shows the code of the `or` macro. It is the simpler variant of the `query/default` macro we saw in chapter 1, in that it ensures that the expression `e1` is evaluated only once, and if it is `false`, the value of `e2` is returned. We introduce a fresh variable x to avoid any unintended captures.

```
fun or (e1 : expr, e2: expr) returns r: expr is
  fresh x in Let(x, e1, If(Var(x),Var(x),e2))
end.
```

Example 5.4 – The or-macro, slightly incomplete

```
fun expand (e : expr) returns r : expr/base where fr(r)=fr(e) is
  case e of
  | Const c => Const/base(c)
  | Var x => Var/base(x)
  | Abs x ex => Abs/base(x,expand(ex))
  | App e1 e2 => App/base(expand(e1),expand(e2))
  | If e1 e2 e3 => If/base(expand(e1),expand(e2),expand(e3))
  | Let x e1 e2 => Let/base(x,expand(e1),expand(e2))
  | Or e1 e2 => expand(or(e1,e2))
  end.
end.
```

There may be an atom that is:
- A free reference in e1
and thus free on the right-hand side, but not on the left-hand side.

In the counterexample, the atom appears as:
- A free reference in e2
- A free reference in e1

Example 5.5 – Something is wrong with the free references

When we check this macro and its expander, we get the error message in Example 5.5. What information do we get from this? Because it is highlighted, it is clear that the one goal clause of the annotated postcondition could not be discharged. The reason for that, according to the error message, is that `e1` may contain a free reference that does not appear in the result (that is why `e1` is marked). The arrow of the or-case is highlighted because it signals the branch where the failed postcondition comes from (in case the responsible variable does not turn up in the respective line). We are also informed that in the generated counterexample, our problematic atom also appears as a free reference in `e2`. That is strange: it means that one or more free references in both arguments to `or` are somehow lost. But all we do at this point are two function calls: one to expand itself—which assures that whatever expression it gets, the free references stay the same, and the call to the `or`-function. A quick inspection tells us that we forgot to write a postcondition for the or-function. Because we want to preserve the free references from the input in the output, that postcondition must be:

$$\mathcal{F}_r(r) = \mathcal{F}_r(e1) \cup \mathcal{F}_r(e2)$$

5.5.2 Binders vs. References

In the letrec-algorithm that we transcribed[1] from Herman [7], we encountered another possible mistake: the algorithm first "unzips" all the let-clauses into two lists—one contains the binders and one contains the value expressions. It also generates a list of equal length that contains `false`-literals—these will be used for the initial values of all the recursively defined variables. At the end of the unzipping process, an n-ary λ-abstraction is generated that uses the accumulated list of binders as its list of formals. Essentially representing a `let`-expression, this λ is immediately applied to the list of `false`-literals. The body of the abstraction is another macro that generates a `set`-expression for every let-clause, followed by the body of the `letrec`.

That other macro again needs a list of atoms to generate a `set`-expression for each of them. In a `set`-expression, we need a `reference` atom. We know that Romeo-L handles the conversion between references and binders automatically as needed, so instead of defining our own type for a list of references, we can try to use the list of free binders as an argument for the macro (called `Begin-Set`).

[1] See the complete transcribed algorithm in Appendix I.

```
fun expand (e : expr) returns r : expr/base where fr(r)=fr(e) is
  case e of
  | Const c => Const/base(c)
  | Var x => Var/base(x)
  | Abs xs ex => Abs/base(xs,expand(ex))
  | App ex exs => App/base(expand(ex),expand/list(exs))
  | Set x ex => Set/base(x,expand(ex))
  | Seq e1 e2 => Seq/base(expand(e1),expand(e2))
  | Letrec lrc ex => expand(letrec(lrc,ex))
  | Letrec/Unzip fs inis acts lrc ex =>
        expand(letrec/unzip(fs inis acts lrc ex))
  | Begin-Set refs acts ex => expand(begin-set(refs acts ex))
  end.
end.
```

There may be an atom that is:

- A free reference in refs, acts, OR ex

and thus free on the left-hand side, but not on the right-hand side.

In the counterexample, the atom appears as:

- A free reference in refs, acts, OR ex

- A free binder in refs

Example 5.6 – An error in an attempt to implement letrec

Example 5.6 shows us the error message we get for trying the above shortcut. Again, the `expand`-function is the first to fail, telling us that it cannot preserve the free references in case it encounters a `Begin-Set`. We are told that the result may contain a free reference that was not in the argument. Because a function may do anything with its arguments and it is not easy to deduce the flow from arbitrary postconditions, we have to mark all arguments to a function call if its result is problematic (as discussed in section 4.3.2). That is not very helpful.

What is helpful, though, is where in the counterexample our problematic atom also appears: as a free binder in `refs`, which is our list of binders. Of course! These get converted into references in the `set`-expressions—we are currently evaluating the body of the λ-abstraction that binds them. The postcondition of the `begin-set`-function correctly states this conversion, but `expand` cannot work with that. There, the free binders in `refs` are actually exposable atoms in `e` and should therefore not be free in the result anyway. The solution in this case is to write a work-around: we introduce a type for lists of references and write a function to convert from a list of binders to a list of references. A postcondition of that function asserts this conversion by saying that the free binders in the argument are the free references in the result. Then, the unzipping macro just has to convert the list in its last step and put it as an argument to `begin-set` in its output. The complete code can be found in Appendix I.

```
fun escape() returns binder is
  fresh x in ladidah(x)
end.

fun ladidah (x: binder) returns binder is
  x
end.
```

The following constraints cannot be simultaneously satisfied:

- Freshness of variable x

- Knowledge after function call

(free atoms in result must be subset of free atoms of arguments)

- Inferred set size:

There is exactly one free binder in the result of the call to ladidah

Example 5.7 – An error related to a fresh-expression

5.5.3 Freshness and Unsatisfiability

In Example 5.7, we try to fool the proof system by generating a fresh variable and then trying to expose it by giving it to an identity function for binders and returning the result of that call. But the proof system caught us. It can even tell us that this is never going to be satisfiable, and explains (slightly re-ordered):

- The atom in x is fresh, so it may not escape.
- We know that there must be a free binder in the result of the function call
- But the free atoms in the result of the function call must be a subset of the free atoms in its arguments (and x is the only one)

```
type Pair is
  | P reference reference
end.

fun incomp1 (r1 : reference, r2 : reference, r3 : reference, p : Pair)
  where fr(r1) # fr(r2) ∧ fr(r1) # fr(r3) ∧ fr(r2) # fr(r3) ∧
        fr(r1) ∪ fr(r2) ∪ fr(r3) ⊆ fr(p)
  returns out : Pair where fr(out) = fr(p) is
    absurd Pair
end.

fun incomp2 (r1 : reference, r2 : reference, p : Pair)
  where fr(r1) # fr(r2) ∧ fr(r1) ⊆  fr(p) ∧ fr(r2) ⊆ fr(p)
  returns out : Pair where fr(out) \ fr(r1) = fr(r2) is
    p
end.
```

Example 5.8 – Incompleteness of Romeo's approximation strategy

- Therefore, the atom in x would always escape, so this program will never work

5.6 Incompleteness

Example 5.8 shows a two functions in a simple Romeo-L program. The proof obligations of the first one are absurd, those of the second one are correct. Both of these statements cannot be proven by the proof system. The reason is the incomplete approximation: in both cases, we do not generate precise enough set size constraints to let the SMT solver know that the number of references in the pair is two.

In the first function, we assert that the union of free references in three distinct atoms is a subset of the free references in the pair. This is absurd, but the proof system cannot prove that—the absurdity assertion is rejected.

Similarly, in the second function, we have a postcondition that will always be satisfied: if the sets of free references in two distinct atoms are both subsets of the free references in a pair, we know that the former must constitute both and therefore all parts of the pair, and removing one of them just leaves the other. The proof system however generates a counter-example that says that the result may contain an additional free atom that comes from the argument p.

This is nevertheless an improvement over pure FreshML's incompleteness, which only considered the difference between empty and non-empty. In fact, the difference is more than just adding one as possibility. Given several size-one-constraints, the SMT-solver can add them up appropriately: in Example 5.8, we could actually correctly determine the size of p and thus prove that the constraints hold if we would deconstruct p with a case expression.

5.7 Gravity

'I like to think of types as warping our gravity, so that the direction we need to travel becomes "downhill".'

Conor McBride, cited by Kiselyov et al. [30]

Sadly, this statement is not true for Romeo(-L). The same written program may be valid with different kinds of binding specifications and constraints. However, the binding specifications affect the semantics of the program, and the constraints affect the guarantees. Simply relaxing the constraints to get a program to be admitted by the proof system may not result in the program one would have wanted, similarly, changing the binding structure may yield a program that is very different from the original one, which may or may not be closer to the goal.

6 Epilogue

6.1 Closing the Loop

Recall the query/default-macro from Example 1.1 that we used to explain problems with naïve macro expansion in section 1.3. Example 6.1 shows how this macro is expressed in Romeo-L when following the style discussed in section 5.3. Romeo will make sure that this function is executed in a hygienic way. We have seen some more examples in the previous chapter, and the Appendix contains the code for two more examples—taken from the literature discussed in chapter 1—which can be checked and are accepted by Romeo(-L).

6.2 Summary

We have analyzed the proof system of Romeo [19] and showed that it can be used in a way that minimizes the additional burden it places on the programmer to write down annotations. We also showed that errors that arise when we are not able to discharge a proof obligation can be explained to the programmer in a meaningful way, by which we mean that we

```
type expr is
  …
  | Query expr
  | Query/Default expr expr
end.

fun query/default(e1 : expr, e2: expr) returns r :expr
 where fr(r) = fr(e1) ∪ fr(e2) is
  fresh x in Let(x,e1,If(Var(x),Query(Var(x)),e2))
end.
```

Example 6.1 – The query/default-macro in Romeo-L can point to related positions in the actual source code and give some background information on where problematic atoms might turn up.

We took these observations and used them to implement Romeo-L, which provides a simpler syntax to write Romeo programs. In the translation process to Romeo, we also collect all the necessary data to provide error reporting.

Working with examples, we saw that the benefits of inferred constraints are huge, and that the complexity of set theory does not seem to affect the running time of the process of discharging proof obligations.

6.3 Future Work

Apart from keeping up with core Romeo, there are various other ways in which Romeo-L could be improved. First, we saw in section 5.3 that the way we model macros is relatively uniform. The next logical step after Romeo-L would be to look at ways to make macro-writing easier, without the boilerplate code for the `expand`-function and the doubled type definitions.

Romeo-L currently does only rudimentary type-checking—mostly to be able to infer conversions between binders and references, and at some places for debugging output. Implementing type-checking there would make it easier to give error messages about typing errors.

Since Romeo-L already generates almost all the constraint information in order to be able to match it with the actual constraints to provide error reports, one could go even further and just let Romeo-L generate the proof obligations on its own. The simplification and approximation procedures would have to be rewritten, too, but in the end, there would be a more uniform system that could generate proof obligations right with their error reporting data included instead of having to match them up later.

Our ability to express cardinality constraints for sets is currently limited because it is not supported by the theory that we are using. However, there are theories that offer cardinality constraints with Presburger Arithmetic [31]. It is unclear whether this extension would give an immediate benefit without also changing the logic behind the proof system, but this path seems worth exploring.

Finally, increasing the “gravity” of the proof system will be a challenging, but worthwhile task. There may at least be heuristics that can produce explanations for common errors that lead programmers into a reasonable direction.

References

[1] B. Pierce, Types and Programming Languages, MIT Press, 2002.

[2] S. Ganz, A. Sabry and W. Taha, "Macros as MultiStage Computations: TypeSafe, Generative, Binding Macros in MacroML," in *ICFP'01 - Proceedings of the Sixth ACM SIGPLAN International Conference on Functional Programming*, Florence, Italy, 2001.

[3] J. Baker and W. C. Hsieh, "Maya: Multiple-Dispatch Syntax Extension in Java," in *PLDI'02 - Proceedings of the 2002 ACM SIGPLAN Conference on Programming Language Design and Implementation*, Berlin, Germany, 2002.

[4] E. E. Kohlbecker, D. P. Friedman, M. Felleisen and B. F. Duba, "Hygienic Macro Expansion," in *LISP and Functional Programming*, 1986, pp. 151-161.

[5] W. D. Clinger and J. Rees, "Macros That Work," in *POPL'91 - Conference Record of the Eighteenth Annual ACM Symposium on Principles of Programming Languages*, Orlando, FL, 1991.

[6] R. K. Dybvig, R. Hieb and C. Bruggeman, "Syntactic Abstraction in Scheme," *Lisp and Symbolic Computation,* vol. 5, no. 4, pp. 295-326, 1992.

[7] D. Herman, A Theory of Typed Hygienic Macros, Dissertation, Northeastern University, 2010.

[8] B. Aktemur, Y. Kameyama, O. Kiselyov and C.-c. Shan, "Shonan Challenge for Generative Programming," in *PEPM'13 - Proceedings of the ACM SIGPLAN 2013 Workshop on Partial Evaluation and Program Manipulation*, Rome, Italy, 2013.

[9] A. Bawden and J. Rees, "Syntactic closures," in *LISP and Functional Programming*, Snowbird, UT, 1988.

[10] D. Herman and M. Wand, "A Theory of Hygienic Macros," in *ESOP'08 - Programming Languages and Systems, 17th European Symposium on Programming, ESOP 2008, Held*

as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, 2008.

[11] N. G. De Bruijn, "Lambda Calculus Notation with Nameless Dummies: A Tool for Automatic Formula Manipulation, with Application to the Church-Rosser Theorem," in *Indagationes Mathematicae*, Elsevier, 1972, pp. 381-392.

[12] M. Gabbay and A. M. Pitts, "A New Approach to Abstract Syntax with Variable Binding," *Formal Aspects of Computing,* vol. 13, no. 3-5, pp. 341-363, 2002.

[13] C. McBride and J. McKinna, "I am not a number: I am a free variable," in *Haskell'04 - Proceedings of the ACM SIGPLAN Workshop on Haskell*, Snowbird, UT, 2004.

[14] M. R. Shinwell, A. M. Pitts and M. Gabbay, "FreshML: Programming with Binders Made Simple," in *ICFP'03 - Proceedings of the Eighth ACM SIGPLAN International Conference on Functional Programming*, Uppsala, Sweden, 2003.

[15] P. Sewell, F. Z. Nardelli, S. Owens, G. Peskine, T. Ridge, S. Sarkar and R. Strnisa, "Ott: Effective tool support for the working semanticist," *Journal of Functional Programming,* vol. 20, no. 1, pp. 71-122, 2010.

[16] S. Weirich, B. A. Yorgey and T. Sheard, "Binders Unbound," in *ICFP'11 - Proceeding of the 16th ACM SIGPLAN International Conference on Functional Programming*, Tokyo, Japan, 2011.

[17] M. R. Shinwell, The Fresh Approach: functional programming with names and binders, Dissertation, University of Cambridge, UK, 2005.

[18] F. Pottier, "Static Name Control for FreshML," in *LICS'07 - 22nd IEEE Symposium on Logic in Computer Science* , Wroclaw, Poland, 2007.

[19] P. Stansifer and M. Wand, "Alpha-Agnostic Programming," In Preparation.

[20] M. Might, Environment Analysis of Higher-Order Languages, Dissertation, Georgia Institute of Technology, 2007.

[21] F. Pottier, "An Overview of Cαml," *Electronic Notes in Theoretical Computer Science,* vol. 148, no. 2, pp. 27-52, 2006.

[22] M. Barnett, B.-Y. E. Chang, R. DeLine, B. Jacobs and K. R. M. Leino, "Boogie: A Modular Reusable Verifier for Object-Oriented Programs," in *FMCO'05 - Formal Methods for*

Components and Objects, 4th International Symposium, Amsterdam, The Netherlands, 2005.

[23] A. R. Bradley, Z. Manna and H. B. Sipma, "What's Decidable About Arrays?," in *VMCAI'06 - Verification, Model Checking, and Abstract Interpretation, 7th International Conference*, Charleston, SC, 2006.

[24] L. De Moura and N. Bjørner, "Generalized, Efficient Array Decisition Procedures," in *FMCAD'09 - Proceedings of 9th International Conference on Formal Methods in Computer-Aided Design*, Austin, TX, 2009.

[25] M. Felleisen, R. B. Findler and M. Flatt, Semantics Engineering with PLT Redex, MIT Press, 2009.

[26] L. De Moura and N. Bjørner, "Z3: An efficient SMT solver," in *TACAS'08 - Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008*, Budapest, Hungary, 2008.

[27] M. Wand, "Finding the Source of Type Errors," in *POPL'86 - Conference Record of the Thirteenth Annual ACM Symposium on Principles of Programming Languages*, St. Petersburg Beach, FL, 1986.

[28] F. Pottier, "Pure FreshML Homepage," 21 03 2012. [Online].
Available: http://pauillac.inria.fr/~fpottier/freshml/freshml.html.en.
[Accessed 15 04 2013].

[29] K. R. M. Leino, M. Moskal and W. Schulte, "Verification Condition Splitting,"
10 10 2009. [Online].
Available: http://research.microsoft.com/apps/pubs/default.aspx?id=77373.
[Accessed 10 04 2013].

[30] O. Kiselyov and C.-c. Shan, "Programming as collaborative reference," in *Off the Beaten Track (POPL)*, Philadelphia, PA, 2012.

[31] P. Suter, R. Steiger and V. Kuncak, "Sets with Cardinality Constraints in Satisfiability Modulo Theories," in *VMCAI'11 - Verification, Model Checking, and Abstract Interpretation - 12th International Conference*, Austin, TX, 2011.

APPENDIX I - ADDITIONAL CODE

1 Letrec

```
;; Translated from the algorithm given by Herman [7]
type const is
  | False
  | Any
end.

type expr is
  | Const const
  | Var reference
  | Abs binder-list expr↓(0)
  | App expr expr-list
  | Set reference expr
  | Seq expr expr
end.

type expr/ext is
  | Const/ext const
  | Var/ext reference
  | Abs/ext binder-list expr/ext↓(0)
  | App/ext expr/ext expr-list/ext
  | Set/ext reference expr/ext
  | Seq/ext expr/ext expr/ext
  | Letrec letrec-clauses expr/ext↓(0)
  | Letrec/Unzip binder-list const-list/ext
                 expr-list/ext letrec-clauses expr/ext↓(0 3)
  | Begin-Set ref-list expr-list/ext expr/ext
end.

type expr-list is
  | ENil | ECons expr expr-list
end.

type expr-list/ext is
  | ENil/ext | ECons/ext expr/ext expr-list/ext
end.

type const-list/ext is
  | CNil/ext | CCons/ext const const-list/ext
end.
```

```
type binder-list is
  | BNil | BCons binder binder-list ⇑(0 1)
end.

type ref-list is
  | RNil | RCons reference ref-list
end.

type letrec-clauses is
  | LRNil | LRCons binder expr/ext↓(0 2) letrec-clauses↓(0) ⇑(0 2)
end.

fun expand (e : expr/ext) returns r : expr
 where fr(r) = fr(e) is
  case e of
  | Const/ext c => Const(c)
  | Var/ext x => Var(x)
  | Abs/ext xs ex => Abs(xs,expand(ex))
  | App/ext ex exs => App(expand(ex),expand/list(exs))
  | Set/ext x ex => Set(x,expand(ex))
  | Seq/ext e1 e2 => Seq(expand(e1),expand(e2))
  | Letrec lrc ex => expand(letrec(lrc,ex))
  | Letrec/Unzip fs inis acts lrc ex =>
      expand(letrec/unzip(fs inis acts lrc ex))
  | Begin-Set refs acts ex => expand(begin-set(refs acts ex))
  end.
end.

fun expand/list (e : expr-list/ext) returns r : expr-list
 where fr(r) = fr(e) is
  case e of
  | ENil/ext => ENil()
  | ECons/ext ex rest => ECons(expand(ex),expand/list(rest))
  end.
end.

fun letrec(lrc : letrec-clauses, e : expr/ext) returns r: expr/ext
 where fr(r) = fr(lrc) ∪ (fr(e) \ fb(lrc)) is
  Letrec/Unzip(BNil(),CNil/ext(),ENil/ext(),lrc,e)
end.
```

```
fun const-to-expr-list (cs : const-list/ext)
 returns es: expr-list/ext where fr(es) = ∅ is
  case cs of
  | CNil/ext => ENil/ext()
  | CCons/ext c rest =>
      ECons/ext(Const/ext(c),const-to-expr-list(rest))
  end.
end.

fun binder-to-ref-list (bs : binder-list) returns rs: ref-list where fr(rs) =
fb(bs) is
  case bs of
  | BNil => RNil()
  | BCons x rest => RCons(x,binder-to-ref-list(rest))
  end.
end.

fun letrec/unzip (formals : binder-list,
                  initials : const-list/ext,
                  actuals : expr-list/ext,
                  clauses : letrec-clauses,
                  body : expr/ext)
  returns r : expr/ext
 where fr(r) = (fr(body)∪ fr(actuals) ∪ fr(initials)∪ fr(clauses))
                    \ (fb(formals) ∪fb(clauses)) is
  case clauses of
  | LRNil =>
     App/ext(Abs/ext(formals,
                     Begin-Set(binder-to-ref-list(formals),
                               actuals,
                               body)),
             const-to-expr-list(initials))
  | LRCons x e rest =>
      letrec/unzip(BCons(x,formals),
                   CCons/ext(False(),initials),
                   ECons/ext(e,actuals),
                   rest, body)
  end.
end.
```

```
fun begin-set
  (formals : ref-list, actuals : expr-list/ext, body : expr/ext)

 returns r : expr/ext
 where fr(r) = fr(body) ∪ fr(actuals) ∪ fr(formals) is
  case formals of
  | RNil => case actuals of
            | ENil/ext => body
            | default => fail expr/ext
            end.
  | RCons x rr =>
    case actuals of
    | ENil/ext => fail expr/ext
    | ECons/ext e er =>
      fresh tmp in
       App/ext(Abs/ext(BCons(tmp,BNil()),
               Begin-Set(rr,er,
                         Seq/ext(Set/ext(x,Var/ext(tmp)),body))),
               ECons/ext(e,ENil/ext()))
    end.
  end.
end.
```

2 Normalization by Evaluation

```
;; Translated from Pottier’s version [18], where it is based on the algorithm
given by Shinwell, Pitts and Gabbay [14].

type lam is
  | Var reference
  | Lam binder lam↓(0)
  | App lam lam
end.

type sem is
  | L env binder lam↓(0 1)
  | N neu
end.

type neu is
  | V reference
  | A neu sem
end.

type env is
  | ENil
  | ECons env binder sem ⇑(0 1)
end.

fun reify (s : sem) returns r : lam is
  case s of
  | L env y t =>
      fresh x in
      Lam(x,reify(evals(ECons(env,y,N(V(x))),t)))
  | N n =>
      reifyn(n)
  end.
end.

fun reifyn (n : neu) returns r: lam
  is
  case n of
  | V x =>
    Var(x)
  | A nn d =>
    App(reifyn(nn),reify(d))
  end.
end.
```

```
fun evals (env : env t : lam) returns v : sem
  where fa(v) ⊆ fr(env) ∪ (fa (t) \ fb(env)) is
  case t of
  | Var x =>
    case env of
    | ENil =>
      N(V(x))
    | ECons tail y v =>
      if x = y then v
      else evals(tail,t)
    end.
  | Lam x tt =>
    L(env x tt)
  | App t1 t2 =>
    case evals(env t1) of
    | L cenv x tt =>
      evals(ECons(cenv,x,evals(env,t2)) tt)
    | N n =>
      N(A(n,evals(env,t2)))
    end.
  end.
end.

fun eval (t : lam) returns sem is
  evals(ENil(),t)
end.

fun normalize (t : lam) returns lam is
  reify(eval(t))
end.
```