Dynamic Scope Analysis for Emacs Lisp

Diplomarbeit im Fach Informatik

an der
Eberhard-Karls-Universität Tübingen

vorgelegt von
Matthias Neubauer

Tübingen, Dezember 2000

Erklärung

Hiermit versichere ich, diese Arbeit selbständig verfaßt und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt zu haben.

Tübingen, den 13. Dezember 2000

Prüfer:	Prof. Dr. H. Klaeren
Betreuer:	Dipl.-Inform. M. Sperber

Zusammenfassung

Dynamic Scope Analysis ist eine statische Programmanalyse für Emacs-Lisp-Programme. Mit ihrer Hilfe können bestehende Emacs-Lisp-Programme in modernere Programmiersprachen mit statischer Bindung übersetzt werden. Die entwickelte Analysetechnik ist eine neue Variante bekannter constraint-basierter Flußanalyse-Techniken, wobei zusätzlich Variablenbindungsorte als zentrales Kriterium zur Verbesserung der Genauigkeit zum Einsatz kommen. Eine Übersetzung kann aus den Informationen der Analyse in mehreren Punkten profitieren: sie ermöglichen in vielen Fällen die sichere Ersetzung dynamischer Bindungen durch lexikalische Bindungen, zeigen Konflikte von Funktions- und Variablennamen auf und erlauben häufig eine Trennung von Code und Daten.

Die Arbeit beinhaltet eine formale Spezifikation der Analyse für eine Teilmenge von Emacs Lisp, formuliert ein Korrektheitsresultat der Analyse bezüglich einer operationellen Reduktionssematik und erläutert ihre Implementierung.

Abstract

Dynamic Scope Analysis is a static analysis for Emacs Lisp programs. It enables the compilation of existing programs written in Emacs Lisp into more modern programming languages using lexical scoping. Dynamic Scope Analysis is a new variant of a class of known analysis techniques called *constraint-based flow analyses*. Its central criteria to gain precision is the usage of *variable birthplaces*. A compilation can benefit in several ways, when based on information gained from this approach: it can substitute dynamic variable bindings by static variable bindings, it makes conflicts between the variable and the function name space obvious, and it often allows a true separation between code and data.

This thesis contains a formal specification of the analysis for a subset of Emacs Lisp and a correctness result for it with respect to the structural operational semantics in the style of Wright and Felleisen with evaluation contexts and rewriting rules. Finally, there is the description of the analysis' implementation in Scheme.

Contents

List of Figures

Chapter 1

Introduction

Emacs Lisp is about to become a dead language. Send it to heaven with Dynamic Scope Analysis!

The Emacs editor with all its derivatives is — despite the fact that it has been around for more than two decades — still one of the most popular companions for the better part of the world's programmers. Surely, one of the main reasons is its extensibility: Emacs has Emacs Lisp [LLStG85, Win99]. Considered from today's perspective, the Emacs Lisp programming language has two major drawbacks: it is an ancient language and its implementations are slow in comparison to more modern programming languages. Emacs Lisp has intrinsic concepts such as the exclusive use of dynamic scoping, the existence of two separate name spaces, and the blurring of code and data. Consequently, the language hinders adequate use abstraction, and in addition, the result is a slow program.

Various attempts have begun to migrate from the existing extension language to a more modern language by substituting the existing interpreter for Emacs Lisp by another for designated new extension languages [Bir93, GUI00, Bot00]. In 1998 a group of researchers and students at the Wilhelm-Schickard-Institut in Tübingen originated the `el->scm` project to replace Emacs Lisp by the Scheme programming language [KCR98] in the Emacs descendant XEmacs [BGHS00, Bib00]. Such a replacement is too big for just one person because of the tremendously huge code base of existing Emacs Lisp programs which must still function after the migration. The plan to attack this problem involves a source-to-source compiler that produces readable and still maintainable Scheme code by an *idiomatic translation*, and also an interpreter for the old Emacs Lisp language.

Comparing the expressive power of programming language, Matthias Felleisen argues that a highly desired property of programming languages would be that "the translation between languages should preserve as much of a program's structure as possible." [Fel91, page 40] The requirement is exactly the same for *idiomatic translation*: the resulting code should keep the appearance of the original code as much as possible. To achieve this, a tool that performs a static program analysis on Emacs Lisp programs is the key which this thesis develops. The tool should obtain additional information about the programs at compile time which help to obtain a reasonable translation.

Several basic approaches for doing static program analysis are known. One way is to use a discipline called *type systems* that adapts or extends a known type system to meet the additional needs. A second approach, *abstract interpretation*, was invented by Patrick and Radhia Cousot [CC77] involves abstracting an existing interpreter to get the desired information. Approaches from a third discipline are called *constraint-based* or *set-based flow analyses*. These methods work in two phases: first, they generate a set of constraints that express the dependencies between the wanted information; then, they solve the constraints to obtain the information.

In order to prove the viability of an analysis there must be a formal way to discuss the dynamic behavior of programs in Emacs Lisp in terms of their formal meanings and semantics.

The first part of the way to the analysis is the presentation of a formal syntax and semantics for a subset of Emacs Lisp, called *Mini Emacs Lisp*. The formalization of Mini Emacs Lisp is based upon a structural operational semantics or small-step semantics as promoted by Matthias Felleisen and Daniel P. Friedman [FF86] in 1986. The second part specifies a static program analysis named *Dynamic Scope Analysis* for Mini Emacs Lisp using a constraint-based approach. This analysis outputs exactly the information needed to build a source-to-source compiler for a more modern programming language. The second part of the thesis also contains the proof of the correctness of the analysis with respect to the semantics developed in the first part. In addition, the entire theoretical analysis has been implemented for almost all of Emacs Lisp — written in Scheme. The third parts describes its design and implementation.

1.1 Overview of the thesis

The rest of chapter 1 gives an overview of Emacs Lisp — mainly describing its origins and its peculiarities. Further, it contains an overview of techniques for specifying the semantics of programming languages, and briefly discusses the different approaches to static program analysis.

Chapter 2 includes the formalization of a subset of Emacs Lisp — the language Mini Emacs Lisp — which is relevant for the purpose of this thesis. Included is a formal syntax and semantics for the language. The semantics is specified as a *structural operational semantics* using *evaluation contexts* and rewriting rules.

Chapter 3 deals with the static program analysis. A constraint based flow analysis, called *Dynamic Scope Analysis*, is presented. The main feature of this analysis is the separation of traced data at program points with regard to the birthplaces of variables. The correctness of the analysis with respect to the semantics of Chapter 2 is stated and proven.

Chapter 4 first describes the methodology of generating and solving constraints for our analysis in an abstract fashion. The implementation of the analysis is described by its different phases.

Chapter 5 concludes the thesis. It delineates the limitations of the chosen approach, gives directions for future work and summarizes the results.

1.2 Emacs Lisp

Richard M. Stallman created the first EMACS editor in the mid-seventies at the Massachusetts Institute of Technology (MIT) [Sta79]. It was based on an existing editor system called TECO which was running at MIT on a PDP-10 computer, under their proprietary operating system called Incompatible Timesharing System (ITS). Stallman extended TECO by a set of programming constructs. He then named it EMACS — Editing MACroS.

Soon thereafter, a great number of Emacs clones for a huge variety of computers cropped up, running on all kinds of different operating systems. Even in the early days, computer scientists recognized that programming with the TECO constructs was not the final answer to the extension problem. Since the Lisp programming language was popular at the time [McC81, SG93] and also seemed a perfect replacement for the TECO constructs, the first Emacs clones with Lisp as the extension language began appearing by the end of the seventies [Gre80]. In the early eighties James Gosling developed the first Emacs port for Unix operating systems [Gos81], also integrating a very restricted Lisp dialect called *Mock Lisp*. Based on Gosling's work, Stallman started in 1983 with his version, baptized GNU Emacs. So sprang Emacs Lisp to life — and since newer versions of GNU Emacs and other clones like XEmacs are still in widespread use and they still use the same old language which has remained unchanged since 1983. For example, in a XEmacs, only the editor core and time-critical parts are written in the C programming language. All the contained language modes, mail readers, games, and so on are all written in Emacs Lisp.

The Lisp programming language — or rather the family of programming languages grouped under the name Lisp — originates even earlier. In 1958 McCarthy noticed and redressed the lack of a decent programming language with its basis in algebraic list processing [McC81]. The first language evolved in many directions. MacLisp, developed at MIT in the seventies, was the first implementation that could be considered as a stable point in the history of Lisp. Later Common Lisp and Scheme emerged on different branches as standardized programming languages. The development of Emacs Lisp was heavily influenced by MacLisp with some traces of Common Lisp [LLStG85, Win99].

What follows are the three major peculiarities of Emacs Lisp which account for the fact that nowadays most sane programmers consider Emacs Lisp an antiquated and slow programming language.

Peculiarity 1 — Dynamic Binding

Emacs Lisp uses *dynamic binding*. A variable in Emacs Lisp always refers to the most recent binding of the variable at runtime. Figure 1.1 gives an example demonstrating dynamic binding.

```
(defvar x 4711)

(defun foo ()
  x)

(let ((x 42))
  (foo))
⟹ 42
```

Figure 1.1: Peculiarity 1 — Dynamic Binding.

The first definition introduces a global variable `x` with initial value `4711`. The second definition introduces a global procedure named `foo`. When called, `foo` returns the value bound to the last variable with the name `x`. A local redefinition of `x` to `42` prior to a call to `foo` returns the last bound value to `x` — `42`.

On rare occasions dynamic variables can be useful — for example, the possibility to dynamically bind i/o ports during program execution. However, they lead to drawbacks most of the time. Their code is not clear to read and code optimizations with respect to variable references are much harder to achieve, because variable reference points are determined at run-time. Nevertheless, in many cases, there is only one point to which a variable refers and that point often coincides with the variable definition one observes under static binding. This analysis will identify such variable bindings, because all newer programming languages, including the target language Scheme, use static binding by default. The translation of those instances is straightforward — and thus nicely *idiomatic*.

Peculiarity 2 — Two Name Spaces

Emacs Lisp knows λ-expressions — that is, anonymous procedures are first-class citizens in Emacs Lisp. They are treated as regular values, and therefore can be passed to and returned from any other procedure, or stored in data structures. Nonetheless, Emacs Lisp distinguishes between two name spaces for variables: one name space is intended for "regular variables," the other one for global procedures. Lisp literature considers Emacs Lisp a member of the Lisp-2 world, whereas Scheme, having only one name space, is a member of the Lisp-1 world [SG93, Que96]. In Emacs Lisp, a variable name refers to one of the two name spaces, depending on the context of its appearance. The analysis for Emacs Lisp must identify those variable names which are used for both name spaces at the same time. Figure 1.2 gives an example for this peculiarity.

```
(defvar foo 42)

(defun foo (x)
  (* x x))

(foo foo)
```
$\Longrightarrow 42^2$

Figure 1.2: Peculiarity 2 — Two Name Spaces.

There are two definitions linked to the name `foo`: the *variable* `foo` is bound the the value `42`, and the *function* `foo` is bound to a procedure that returns the square of its argument. When applying `foo` on `foo`, it is clear from the context, that the `foo` in the operator position denotes the function, and the operand `foo` denotes the variable.

Peculiarity 3 — No Distinction Between Code and Data

One general characteristic of Lisp is that every programs has the form of either an atom or a nested lists of atoms. Such expressions are called *S-expressions*. Programs correspond to external representations of the built-in data structures. Thus the creation or manipulation of program code by other programs is very easy. Figure 1.3 shows some examples for that peculiarity.

The first definition introduces the global procedure `foo-maker`, that generates *lists* that look like anonymous procedures. A call to `foo-maker` — here executed by `funcall` — generates such a list. The resulting list not only looks like a procedure, it can also be used as one; the second `funcall` shows the application of this "list". Note, that the first `funcall` does not really call a function name, but the symbol `foo-maker`. As above, symbols both represent symbols and variable names.

It is able that the two entities — programs versus data (i.e. symbols and lists) — are held separately, and transition points are made explicit during a program execution. For both semantical and performance reasons Scheme

```
(defun foo-maker (n)
  (list 'lambda (list 'x) (list '+ 'x n)))

(funcall 'foo-maker 42)
⟹ (lambda (x) (+ x 42))

(funcall (foo-maker 42) 17)
⟹ 59
```

Figure 1.3: Peculiarity 3 — No Distinction Between Code and Data.

has this property, whereas Emacs Lisp makes no distinction between code and data. A static analysis tool for Emacs Lisp should successfully separate objects into code and data.

1.3 Semantics

There are several different approaches to express the semantics of a programming language. Each such approach uses its own formal mathematical language or — more precisely — theory to form the descripting *metalanguage*. A brief overview of the main forms follows.

Denotational Semantics

The goal of this approach dates back to works from Dana S. Scott and Christopher Strachey: describe the different constructs of a programming language in the metalanguage by mathematical objects in a *compositional* fashion. The meaning of an expression is explained by a composition of independent parts of the meanings of its sub-expressions. When describing realistic programming languages — languages which allow general recursion — a metalanguage working with sets is not enough. Objects with more structure are necessary. For this purposes, the theory of *domains* was invented [Mos90, GS90].

Operational Semantics

The category of operational semantics incorporates a whole class of semantical approaches. All these approaches try to describe a language by formalizing the evaluation processes using *abstract machines* — very simple models for real machines which only have the relevant features for the semantics.

Gordon D. Plotkin [Plo81] introduced the *structural operational semantics* known as *small-step semantics*. The evaluation process is subdivided into many small transition steps. Following work by Matthias Felleisen and

Daniel P. Friedman [FF86] expressed this style of semantics as term rewriting systems embedded in evaluation contexts. This is also known as *reduction semantics*.

Gilles Kahn [Kah87] presented a modification of this principal abstracting over smaller steps of the evaluation process, that only a relation of the actual state and the resulting state of a computation is of concern. This form is also known as a *big-step* or *natural semantics*.

Other Approaches to Semantics

Both denotational and operational semantics have a severe drawback: It is nearly impossible to extend the semantics by new concepts without having to rewrite the whole semantics from scratch; a new concept affects the semantics everywhere. Such semantics cannot be considered modular. Modular versions of both denotational and operational semantics remedy these dysfunctionalities. Unfortunately, broad experience how on to use them is still lacking [LHJ95, Mos99].

Other ways to give a program formal meaning such as axiomatic or algebraic semantics are irrelevant for our purposes.

1.4 Static Program Analysis

The ultimate goal of this thesis is to determine certain information from Emacs Lisp programs to support a source-to-source translation using static program analysis. The general idea of such an analysis is to determine prior to a program run — that is, statically at compile time — a safe approximation of all values, types, properties or effects that can occur at interesting program points or for interesting variables.

A static analysis is only able to determine safe approximations of the ultimately required properties, due to the fact that programming languages are in general not decidable. Consider, for example, the simple Scheme program in Figure 1.4.

```
(define (mean-computation x)
  (let ((y Ω))
    x))

(if (> z 0)
    1
    (mean-computation 2))
```

Figure 1.4: The Need of safe Approximations.

The procedure `mean-computation` has one argument. It first evaluates an expression Ω before returning the passed argument. If Ω never terminates, the whole procedure will never terminate. The `if` expression returns either `1` or the result of a call to `mean-computation` with argument `2`. If an analysis wants to determine all values that are possible results of the `if` expression, it can often only give an approximation of the exact result $\{1\}$. The reason is: it is in general not decidable, whether an expression Ω halts always or not. On the other hand, safe approximations would be the sets $\{1, 2\}$ or $\{1, 2, 42\}$. They both contain at least all possibly occurring values. This gives a hint for another property desired for an analysis: the safe approximations should be as close as possible to the ultimately required properties.

There are a wide range of applications for information gained by static program analyses. The classic application is the usage in a compiler. The right information can enable the avoidance of redundant computations or the avoidance of redundant type checks. Other and newer applications are security analyses of software or the checks of software's certain data representations. Prominent examples are tools to reveal Y2K or Euro bugs [EHM+99, RFT99].

The developer can choose between three basic approaches to analyze a functional programming language. The rest of this chapter gives a succinct overview of those mainly inspired by the book of Flemming Nielson, Hanne Riis Nielson and Chris Hankin [NNH99].

Abstract Interpretation

In abstract interpretation [CC77, Myc81], an interpreter executes programs to "compute" the wanted properties instead of the real results. Such an abstract interpreter is obtained by abstracting over its semantic domains such that it propagates only the needed information. Choosing the right semantical domains the abstract interpreter will always terminate. The correctness of such an analysis is generally obtained by showing the correct relation of the real and the abstract world before and after each interpretation step.

Type and Effect Systems

With a type system for a programming language on hand, a developer can extend it to trace additional properties or effects. Usually, if effects of computations are the properties to be traced, the function types will be annotated by effects. Annotated type systems on the other hand trace additional informations for base types: they annotate them with extra information.

Correctness results for these kind of analyses are adaptions or extensions of the correctness results for the underlying type systems. Likewise, the analysis algorithms are adaptions of the original type inference algorithms.

Constraint-Based Analysis

Constraint-based analyses are also known as *set-based analyses*. Heralding figures in this area were Olin Shivers, with the work for his dissertation about flow analyses for Scheme [Shi88, Shi91], and Nevin Heintze, with his results for ML [Hei94].

The general methodology is first to identify the needed level of abstraction for the traced data resulting in the domains of abstract values, abstract environments, and any other objects needed. An analysis, in this sense, is a mapping from program points or variables to sets of abstract values. When developing such an analysis, the first step is usually to specify which analyses really make sense for the language and the problem. Imagine this as stating how the flow of data and control works between the different types of expressions. A correctness result is obtained in a similar fashion as with type systems.

In general, the actual determination of a valid analysis — hopefully also of a "small analysis" — happens in two phases. First, the analysis is described by generated constraints stating the relations between the sets of abstract values which occurs. Second, the generated constraints must be solved to actually get the solution.

There are several known techniques to make a constraint-based analysis more precise. An important factor is the way the traced abstract values are chosen. Tracing less abstract data makes the analysis more precise, but also slower. Another common way is to discriminate the analyzed program points by different additional properties. Shivers for example proposed the differentiation of program points with respect to the locations of the last k happened procedure calls and calls the resulting analysis k-cfa.

In our case, the analysis for Emacs Lisp is a constraint-based approach, discriminating the program points by a new variant: the associated sets of abstract values are split depending on the behavior of the dynamic binding. The semantics is an operational semantics in a style called *reduction semantics*.

Chapter 2

Mini Emacs Lisp

The first aim of this thesis is to provide the practical application of the program analysis for Emacs Lisp with a sound formal basis. The first step in this direction is the formalization of both the look and feel of Emacs Lisp programs. In other words, first a formal abstract syntax specifying valid programs of a language called *Mini Emacs Lisp*, a substantial subset of Emacs Lisp, must be expressed and then a semantics for them must be derived.

2.1 Syntax of Mini Emacs Lisp

The source language is therefore a λ-calculus-like language with additional basic constants and symbols. In contrast to newer Lisp dialects such as Scheme, is the occurrence of two separate kinds of binding, one usually used for function and the other used for variable bindings. In addition, the latter enables the introduction of new bindings using two techniques, a mechanism known as *dynamic binding* and the mutation of existing ones. Also included are a construct for exception handling and a small set of primitives allowing the use of immutable pairs. All program fragments have been provided with labels in order to be able to connect the language with a flow analysis.

Figure 2.1 shows the abstract syntax for expressions (or labeled terms) of Mini Emacs Lisp and the additional elements needed to express them using EBNF style for grammar. The syntactic category **Lab** contains the *labels* which are associated with the terms to create expressions. This category is intentionally left unspecified. **SymVar** qualifies all the *symbols* and at the same time the *variable names* of the languages. They fall together in Emacs Lisp rather than being separated, as in most other languages. Additional *constants* found in **Const** are specified here as natural numbers, but could be easily extended. There is no special syntactic category to express boolean values: Emacs Lisp follows the old Lisp tradition and uses the special symbol `nil` for false and every other value for true. The set of

$$
\begin{array}{rclcl}
l & \in & \textbf{Lab} & ::= & \ldots \\
s, x & \in & \textbf{SymVar} & ::= & \texttt{fritz} \mid \texttt{franz} \mid \ldots \\
c & \in & \textbf{Const} & ::= & \texttt{0} \mid \texttt{1} \mid \texttt{2} \mid \ldots \\
b & \in & \textbf{Prim} & ::= & \texttt{funcall} \mid \texttt{cons} \mid \texttt{car} \mid \texttt{throw} \mid \ldots \\
\\
t & \in & \textbf{Term} & ::= & c \\
 & & & \mid & \texttt{(quote } s\texttt{)} \\
 & & & \mid & \texttt{(lambda (}x\texttt{) } e\texttt{)} \\
 & & & \mid & x \\
 & & & \mid & \texttt{(setq } x\ e\texttt{)} \\
 & & & \mid & \texttt{(}x_0^{l_0}\ e_1\texttt{)} \\
 & & & \mid & \texttt{(let } x\ e_1\ e_2\texttt{)} \\
 & & & \mid & \texttt{(if } e_0\ e_1\ e_2\texttt{)} \\
 & & & \mid & \texttt{(catch } e_0\ e_1\texttt{)} \\
 & & & \mid & \texttt{(}b\ e_1 \ldots e_n\texttt{)} \\
\\
e & \in & \textbf{Exp} & ::= & t^l
\end{array}
$$

Figure 2.1: Syntax of the Expressions in Mini Emacs Lisp.

primitives or built-in procedures **Prim** allows the application of anonymous procedures with `funcall`, the management of pairs with `cons` and `car`, and the throwing of exceptions with `throw`.

The unlabeled expressions, called *terms* **Term**, consist of basic constants, quoted symbols, and λ-terms which construct anonymous procedures. To keep the formalism simple, only procedures with a single argument are used. A single symbol references a variable, whereas in an application, the first symbol references a value from a second name space designed principally to hold procedures. Conditional evaluation is provided by `if`, local dynamically scoped bindings by `let`, and the mutation of the latest binding of a variable by `setq`. Terms beginning with `catch` are constructs to handle exceptions and those beginning with one of the symbols p are calls to primitives. If a term is labeled, it is called an *expression* of **Exp**.

$$
\begin{array}{rclcl}
d & \in & \textbf{Def} & ::= & \texttt{(defvar } x\ e\texttt{)} \\
 & & & \mid & \texttt{(defun } x_0\ \texttt{(}x_1\texttt{) } e\texttt{)} \\
\\
p & \in & \textbf{Prg} & ::= & d^*\ e
\end{array}
$$

Figure 2.2: Syntax of Programs in Mini Emacs Lisp.

Whole programs incooperate top-level definitions of variables and procedures. As seen in Figure 2.2, *definitions* beginning with `defvar` introduce new variables associating initial values, and those beginning with `defun` define new procedures and associate their name in the function name space with procedures. To keep the presentation simple, all procedures have just one parameter. A *program* a of **Prg** is one expression e representing the start of the program execution and also its final value preceded by a sequence of definitions.

Note that there is no need to define the notions of free and bound variables, since Mini Emacs Lisp features dynamic scoping rules. As a consequence, the notion of closed expressions — i.e. expressions without free variables — is not used for the definition of programs.

2.2 Semantics

To be able to formally discuss and reason out the dynamic behavior of programs, it is necessary to state a formal semantics for Mini Emacs Lisp. The semantical properties of Mini Emacs Lisp are expressed as a structural operational or small-step semantics [Plo81]. There are several popular techniques for the formulation of semantics, operational semantics being one of them. Structural operational semantics is one form of operational semantics, for which different styles of presentations exist. An introductory overview can for example be found in [Pit00].

More precisely, the semantics is in a setting featuring *evaluation contexts* and *syntactic rewriting* developed by Felleisen and Friedman [FF86]. The handling of exceptions is inspired by works of Wright and Felleisen [WF91, WF94] and Harper and Stone [HS97].

2.2.1 Values and Intermediate Terms

The representation of the semantics using syntactical means requires the introduction of further syntactic categories for intermediate expressions, values and exceptions as shown in Figure 2.3.

The elements of **Val**, called *values*, are those objects that result from a successful computation. They represent constants, symbols and λ-terms and correspond to the similar expressions of **Exp** which produce them. To make values clearly distinguishable from expressions, the prefixes are different. Objects beginning with `exc` represent *exceptions*. They have two values incorporated; the first one serves as a tag identifying the exception, the second one carries an alternative return value that is used if the exception is handled later during the evaluation. There is only one concept missing: while evaluating the body of a `let` or of a procedure, a variable can temporarily get a new binding.

Neither environments nor local variable substitutions are used for the local binding, instead the *intermediate terms* beginning with `bind` represent the bindings. Again, the values and the intermediate terms each get labels and thus also become members of the set of all expressions.

$$
\begin{array}{rclcl}
f & \in & \textbf{Fun} & ::= & \texttt{(proc (}x\texttt{)}\ e\texttt{)} \\[1ex]
v & \in & \textbf{Val} & ::= & \texttt{(const}\ c\texttt{)} \\
 & & & | & \texttt{(sym}\ s\texttt{)} \\
 & & & | & f \\
 & & & | & \texttt{(pair}\ v_1^{l_1}\ v_2^{l_2}\texttt{)} \\[1ex]
w & \in & \textbf{Exc} & ::= & \texttt{(exc}\ v_0^{l_0}\ v_1^{l_1}\texttt{)} \\[1ex]
it & \in & \textbf{ITerm} & ::= & \texttt{(bind}\ x\ v\ e\texttt{)} \\[1ex]
e & \in & \textbf{Exp} & ::= & v^l \\
 & & & | & w^l \\
 & & & | & it^l
\end{array}
$$

Figure 2.3: Syntax of Values and Intermediate Terms.

2.2.2 Environments

The entities ρ and φ of **Env** describe both kinds of global environments that occur during a program's evaluation. For convenience, the meta-variable ρ always identifies a *variable environment*, whereas a φ identifies a global *function environment*. Structurally speaking they all look the same. Environments are finite mappings from symbols to values

$$\rho, \varphi \in \textbf{Env} = \textbf{SymVar} \rightarrow_{\text{fin}} \textbf{Val}.$$

The presentation of the empty environment is $[]$. The modification of an existing environment through the new mapping of a symbol x to a value v is written as $\rho[x \mapsto v]$.

2.2.3 Evaluation Contexts

Following Barendregt's definition for the λ-calculus [Bar84], *contexts* $\mathcal{C}$ are terms or expressions containing a hole $[-]$ in place of one of its sub-terms or sub-expressions. The notation $\mathcal{C}[e]$ expresses the placement of an expression e in the hole $[-]$ of the context $\mathcal{C}$. Felleisen and Friedman invented the notion of *evaluation contexts* [FF86] in order to specify the reduction strategy of a programming language. In this setting, certain contexts of all conceivable contexts of a programming language are distinguished, in order to clearly denote those positions of every expression where the next redex of the evaluation process must be found. The hole, on the other hand, stands for the place where a rewriting rule for this sub-expression must match, representing one step of evaluation. The evaluation contexts for Mini Emacs Lisp are shown in Figure 2.10.

$$
\begin{array}{rcllll}
\mathcal{E} & \in & \textbf{EvalContext} & ::= & [-] \\
 & & & | & (\texttt{if}\ \mathcal{E}\ e_1\ e_2)^l \\
 & & & | & (\texttt{let}\ x\ \mathcal{E}\ e_2)^l \\
 & & & | & (\texttt{bind}\ x\ v\ \mathcal{E})^l \\
 & & & | & (\texttt{setq}\ x\ \mathcal{E})^l \\
 & & & | & (p\ v^*\ \mathcal{E}\ e^*)^l \\
 & & & | & (\texttt{catch}\ \mathcal{E}\ e_1)^l \\
 & & & | & (\texttt{catch}\ v_0\ \mathcal{E})^l \\
\\
\mathcal{V}_{x_0} & \in & \textbf{VarContext}(x_0) & ::= & [-] \\
 & & & | & (\texttt{if}\ \mathcal{V}_{x_0}\ e_1\ e_2)^l \\
 & & & | & (\texttt{let}\ x\ \mathcal{V}_{x_0}\ e_2)^l \\
 & & & | & (\texttt{bind}\ x_1\ v\ \mathcal{V}_{x_0})^l & \text{if } x_0 \neq x_1 \\
 & & & | & (\texttt{setq}\ x\ \mathcal{V}_{x_0})^l \\
 & & & | & (p\ v^*\ \mathcal{V}_{x_0}\ e^*)^l \\
 & & & | & (\texttt{catch}\ \mathcal{V}_{x_0}\ e_1)^l \\
 & & & | & (\texttt{catch}\ v_0\ \mathcal{V}_{x_0})^l \\
\\
\mathcal{X} & \in & \textbf{ExcContext} & ::= & [-] \\
 & & & | & (\texttt{if}\ \mathcal{X}\ e_1\ e_2)^l \\
 & & & | & (\texttt{let}\ x\ \mathcal{X}\ e_2)^l \\
 & & & | & (\texttt{bind}\ x\ v\ \mathcal{X})^l \\
 & & & | & (\texttt{setq}\ x\ \mathcal{X})^l \\
 & & & | & (p\ v^*\ \mathcal{X}\ e^*)^l \\
 & & & | & (\texttt{catch}\ \mathcal{X}\ e_1)^l
\end{array}
$$

Figure 2.4: Evaluation Contexts.

The grammar of **EvalContext** describes all the *evaluation contexts* in which a reduction step in Mini Emacs Lisp can occurs. For example, as shown by the contexts for `if` expressions, when evaluating such an expression, the evaluation process first completely evaluates the conditional sub-expression, i.e. the first component. To achieve the formulation of the reductions for the references and mutations of a variable correctly, one has to gain access to the last dynamic binding of the variable. The *variable contexts* in $\mathbf{VarContext}(x_0)$ help accomplish this; they describe all contexts without any bindings associated to the symbol x_0. The same concept is needed to handle `catch` expressions smoothly. If an exception occurs, it must be possible to jump out of the context directly towards the innermost `catch`-expression. The *exception contexts* of **ExcContext** provide exactly those contexts that do not contain any handling `catch` expressions.

2.2.4 Reductions

An actual state during the evaluation process is described by a partially evaluated expression and two global environments. Thus, a *configuration* γ of **Conf** is a triple consisting of two environments — the global variable and the global function environment — and the current expression:

$$\gamma \in \mathbf{Conf} = \mathbf{Env} \times \mathbf{Env} \times \mathbf{Exp}.$$

The *reduction rules* of Figures 2.5, 2.6, 2.7, 2.8, 2.9, and 2.11 are the primitive steps of the evaluation process; they show how a certain configuration is rewritten by another.

[**c**]	$\rho, \varphi, \mathcal{E}[c^l] \to \rho, \varphi, \mathcal{E}[\texttt{(const }c\texttt{)}^l]$
[**quote**]	$\rho, \varphi, \mathcal{E}[\texttt{(quote }s\texttt{)}^l] \to \rho, \varphi, \mathcal{E}[\texttt{(sym }s\texttt{)}^l]$
[**lambda**]	$\rho, \varphi, \mathcal{E}[\texttt{(lambda (}x\texttt{) }e\texttt{)}^l] \to \rho, \varphi, \mathcal{E}[\texttt{(proc (}x\texttt{) }e\texttt{)}^l]$

Figure 2.5: Reductions — Value Expressions.

Certain sub-expressions immediately reduce to a value, such as in the rules [**c**], [**quote**], and [**lambda**]. Note that in [**lambda**], it is unnecessary to put all bindings of the procedure in a closure, because the language is a dynamically scoped language. As a result, only the most recent binding of a variable will be of interest. The static bindings, that exist at the defining point of the procedure, are thus unnecessary to remember for the later execution.

A variable is referenced in Mini Emacs Lisp by writing down its symbol (or name). There are two ways — depending on the context in which the

$$[\mathbf{var}] \quad \rho,\varphi,\mathcal{E}[\texttt{(bind}\ x\ v\ \mathcal{V}_x[x^l]\texttt{)}^{l_0}] \rightarrow \rho,\varphi,\mathcal{E}[\texttt{(bind}\ x\ v\ \mathcal{V}_x[v^l]\texttt{)}^{l_0}]$$

$$[\mathbf{var_{glob}}] \quad \rho,\varphi,\mathcal{V}_x[x^l] \rightarrow \rho,\varphi,\mathcal{V}_x[v^l] \quad \text{if } x \in dom(\rho),\ \rho(x) = v$$

$$[\mathbf{setq}] \quad \rho,\varphi,\mathcal{E}[\texttt{(bind}\ x\ v\ \mathcal{V}_x[\texttt{(setq}\ x\ v_0^{l_0}\texttt{)}^l]\texttt{)}^{l_1}] \rightarrow \\ \rho,\varphi,\mathcal{E}[\texttt{(bind}\ x\ v_0\ \mathcal{V}_x[v_0^l]\texttt{)}^{l_1}]$$

$$[\mathbf{setq_{new}}] \quad \rho,\varphi,\mathcal{V}_x[\texttt{(setq}\ x\ v_0^{l_0}\texttt{)}^l] \rightarrow \rho[x \mapsto v_0],\varphi,\mathcal{V}_x[v_0^l]$$

Figure 2.6: Reductions — Variable Bindings.

reference occurs — to determine the bound value of a symbol. If the latest binding occurred locally — that is, a procedure call or a `let`-expression introduced it — the closest `bind`-context for the same variable will hold the associated value. The [**var**]-rule expresses this behavior, whereas the context $\mathcal{V}_x$ guarantees that there is no closer binding to the same variable. If there is no local binding to the given symbol, there can still be a global one. The rule [$\mathbf{var_{glob}}$] tells how to proceed then; the global environment ρ determines whether the rule matches, depending on the containment of an associated value for the symbol x. If this is the case, the found value substitutes x in the hole of the expression. The machinery to mutate bindings by `setq` is analogous to the one to reference variables: In addition, to substitute a value, `setq` also causes the binding in the context to change. In the case of a local binding, the [**setq**] rule changes the value in the corresponding `bind` context. In the case of a global binding or no at all, the global environment is either updated or extended subsequent to [$\mathbf{setq_{new}}$].

$$[\mathbf{app_{lam}}] \quad \rho,\varphi,\mathcal{E}[\texttt{(}x_0^{l_0}\ e_1\texttt{)}^l] \rightarrow \rho,\varphi,\mathcal{E}[\texttt{(let}\ x_1\ e_1\ e_2\texttt{)}^l] \\ \text{if } x_0 \in dom(\varphi),\ \varphi(x_0) = \texttt{(proc (}x_1\texttt{)}\ e_2\texttt{)}$$

$$[\mathbf{app_{ind}}] \quad \rho,\varphi,\mathcal{E}[\texttt{(}x_0^{l_0}\ e_1\texttt{)}^l] \rightarrow \rho,\varphi,\mathcal{E}[\texttt{(}s^{l_0}\ e_1\texttt{)}^l] \\ \text{if } x_0 \in dom(\varphi),\ \varphi(x_0) = \texttt{(sym}\ s\texttt{)}$$

$$[\mathbf{let}] \quad \rho,\varphi,\mathcal{E}[\texttt{(let}\ x\ v_1^{l_1}\ e_2\texttt{)}^l] \rightarrow \rho,\varphi,\mathcal{E}[\texttt{(bind}\ x\ v_1\ e_2\texttt{)}^l]$$

$$[\mathbf{bind}] \quad \rho,\varphi,\mathcal{E}[\texttt{(bind}\ x\ v_0\ v_1^{l_1}\texttt{)}^l] \rightarrow \rho,\varphi,\mathcal{E}[v_1^l]$$

Figure 2.7: Reductions — Procedure Application and Local Variables.

A direct procedure call in Mini Emacs Lisp is only valid if its operator is a symbol. Depending on the value that is bound to that symbol in the function environment — the second component φ of the current configuration — different rewriting rules match: either [$\mathbf{app_{lam}}$] or [$\mathbf{app_{ind}}$].

If the matching rule is [$\mathbf{app_{lam}}$] and the environment φ contains a regular procedure for the symbol x_0, a `let` expression will be the replacement for the original application. The body of the encountered procedure becomes the body of the new `let` expression, and the operator of the application becomes the expression linked to the newly introduced local variable x_1. The [**let**] rule of **EvalContext** guarantees that future evaluation continues on the sub-expression e_2 until it becomes a value (because the language can be considered as a modification of Plotkin's call-by-value λ-calculus). The rule [**let**] rewrites the resulting `let` expression to a similar-looking `bind` expression to start the final evaluation of the body under the new binding. The distinction of `let` expressions and `bind` expressions is inelegant and unnecessary when considering only the semantics, but the correct formulation of the flow analysis enforces their separation. If finally the evaluation of the body under the new binding results in a value, the rule [**bind**] will remove the obsolete context.

The second possibility [$\mathbf{app_{ind}}$], called *function indirection* in the Emacs Lisp world, occurs if the environment φ associates the symbol x_0 with another symbol instead of a procedure. In this case, the application reoccurs in using the new found symbol as the operator of the call instead of the original one.

[$\mathbf{if_1}$]	$\rho, \varphi, \mathcal{E}[\texttt{(if}\ v^{l_0}\ t_1^{l_1}\ e_2\texttt{)}^l] \rightarrow \rho, \varphi, \mathcal{E}[t_1^l]$ if $v \neq \texttt{(sym nil)}$
[$\mathbf{if_2}$]	$\rho, \varphi, \mathcal{E}[\texttt{(if}\ v^{l_0}\ e_2\ t_2^{l_2}\texttt{)}^l] \rightarrow \rho, \varphi, \mathcal{E}[t_2^l]$ if $v = \texttt{(sym nil)}$
[$\mathbf{catch_1}$]	$\rho, \varphi, \mathcal{E}[\texttt{(catch}\ v_0^{l_0}\ v_1^{l_1}\texttt{)}^l] \rightarrow \rho, \varphi, \mathcal{E}[v_1^l]$
[$\mathbf{catch_2}$]	$\rho, \varphi, \mathcal{E}[\texttt{(catch}\ v_t^{l_t}\ \mathcal{X}[\texttt{(exc}\ v_0^{l_0}\ v_1^{l_1}\texttt{)}^{l_2}]\texttt{)}^l] \rightarrow \rho, \varphi, \mathcal{E}[v_1^l]$ if $v_t = v_0$
[$\mathbf{catch_3}$]	$\rho, \varphi, \mathcal{E}[\texttt{(catch}\ v_t^{l_t}\ \mathcal{X}[\texttt{(exc}\ v_0^{l_0}\ v_1^{l_1}\texttt{)}^{l_2}]\texttt{)}^l] \rightarrow \rho, \varphi, \mathcal{E}[\texttt{(exc}\ v_0^{l_0}\ v_1^{l_1}\texttt{)}^l]$ if $v_t \neq v_0$
[**nocatch**]	$\rho, \varphi, \mathcal{X}[\texttt{(exc}\ v_0^{l_0}\ v_1^{l_1}\texttt{)}^{l_2}] \rightarrow \rho, \varphi, \texttt{(exc}\ v_0^{l_0}\ v_1^{l_1}\texttt{)}^{l_3}$ where l_3 is the label of $\mathcal{X}[\texttt{(exc}\ v_0^{l_0}\ v_1^{l_1}\texttt{)}^{l_2}]$

Figure 2.8: Reductions — Conditionals and Exception Handling.

Mini Emacs Lisp features two further control structures. If the evaluation resides in an `if` sub-expression and the conditional component e_0 already is a value, then the whole sub-expression reduces either to the second or the third component, depending on the conditional value (as shown in [$\mathbf{if_1}$] and [$\mathbf{if_2}$] respectively). In the Lisp world, it is uncommon to distinguish between boolean values; it is only important whether the conditional value corresponds to the special symbol `nil` or not.

The rules for a `catch` expression are as follows: According to the evaluation contexts, the evaluation process proceeds with the body of the `catch` sub-expression — in our case the second component e_1. If everything goes well and the body evaluates to a value, [**catch₁**] just passes this value along. Otherwise, in the case that an exception occurs during the process of evaluating the body, either [$\mathbf{catch_2}$] or [$\mathbf{catch_3}$] will match. The surrounding context $\mathcal{X}$ of **ExcContext** symbolizes the sub-expressions to be skipped, that do not contain any closer `catch` blocks. Depending on whether the tag of the `catch` block v_t is textually equal to the tag of the exception v_o or not, either the [$\mathbf{catch_2}$] rule puts the alternative return value provided by the exception in the hole instead of the `catch`-expression or the [$\mathbf{catch_2}$] rule propagates the exception further. The last case — the [**nocatch**] rule — happens, when no `catch` block at all exists to handle the occurring exception. The exception then jumps on the top level.

[**cons**]	$\rho,\varphi,\mathcal{E}[\texttt{(cons } v_1^{l_1}\ v_2^{l_2}\texttt{)}^l] \to \rho,\varphi,\mathcal{E}[\texttt{(pair } v_1^{l_1}\ v_2^{l_2}\texttt{)}^l]$
[$\mathbf{car_1}$]	$\rho,\varphi,\mathcal{E}[\texttt{(car (pair } v_1^{l_1}\ v_2^{l_2}\texttt{)}^{l_0}\texttt{)}^l] \to \rho,\varphi,\mathcal{E}[v_1^l]$
[$\mathbf{car_2}$]	$\rho,\varphi,\mathcal{E}[\texttt{(car (sym nil)}^{l_0}\texttt{)}^l] \to \rho,\varphi,\mathcal{E}[\texttt{(sym nil)}^l]$
[$\mathbf{funcall_{lam}}$]	$\rho,\varphi,\mathcal{E}[\texttt{(funcall (proc (}x_1\texttt{) } e\texttt{)}^{l_0}\ v_1^{l_1}\texttt{)}^l] \to$ $\rho,\varphi,\mathcal{E}[\texttt{(bind } x_1\ v_1\ e\texttt{)}^l]$
[$\mathbf{funcall_{ind}}$]	$\rho,\varphi,\mathcal{E}[\texttt{(funcall (sym } s_1\texttt{)}^{l_0}\ v_1^{l_1}\texttt{)}^l] \to$ $\rho,\varphi,\mathcal{E}[\texttt{(funcall } v_0^{l_0}\ v_1^{l_1}\texttt{)}^l]$ if $s_1 \in dom(\varphi)$, $\varphi(s_1) = v_0$
[**throw**]	$\rho,\varphi,\mathcal{E}[\texttt{(throw } v_1^{l_1}\ v_2^{l_2}\texttt{)}^l] \to \rho,\varphi,\mathcal{E}[\texttt{(exc } v_1^{l_1}\ v_2^{l_2}\texttt{)}^l]$

Figure 2.9: Reductions — Primitives.

Primitives can handle pairs of values completely; new syntactic forms are unnecessary for this purpose. Primitives are built-in functions which the language cannot express itself, rather their semantics has to be stated explicitly. For pairs, `cons` does the construction of pairs by rule [**cons**] and `car` selects the first component of pairs by rule [$\mathbf{car_1}$]. The rule [$\mathbf{car_2}$] corresponds to the special action of `car` on the symbol `nil` in Emacs Lisp. The pairs are integrated to give an example of how to handle aggregates.

The primitive `funcall` offers a more general mechanism to call procedures; The first argument of the primitive must either evaluate to an anonymous procedure or give a symbol for an function environment's entry. The primitive actually applies a procedure to a given operand by reducing the procedure to its body equipped with a new binding for the argument.

The rule [**funcall$_{\mathbf{lam}}$**] treats the case of anonymous procedures. *Function indirection* is handled by the rule [**funcall$_{\mathbf{ind}}$**] accordingly.

The rule [**`throw`**] translates the primitive `throw` to the resembling exception. The effect is to jump out of the current context of evaluation until the closest catching `catch`-expression is reached.

The combination of all the rules from Figure 2.5 to 2.9 defines the reduction relation

$$\rightarrow \subseteq \mathbf{Conf} \times \mathbf{Conf}$$

putting all possible configurations before and after a reduction step during evaluation in relation. Its reflexive, transitive closure is written as $\rightarrow^\star$.

2.2.5 Expression Contexts

So far, only the meaning of expression is defined by the relation $\rightarrow$. The notion of whole programs exists, but there is still no way to interpret programs. For this purpose, consider another kind of contexts, the *expression contexts* of **ExpContext**, as shown in Figure 2.10.

$$\begin{array}{lcll} \mathcal{P} & \in & \mathbf{ExpContext} & ::= \; \texttt{(defvar } x \; [-]\texttt{)} \; p \\ & & & \;\;\;\;\; | \;\;\; [-] \end{array}$$

Figure 2.10: Expression Contexts.

Here, the hole does not stand for the position of the next redex, but rather for the position of the expression which must be evaluated next.

2.2.6 Reductions for Programs

With the adaption of configurations to *program configurations*

$$\delta \in \mathbf{PConf} = \mathbf{Env} \times \mathbf{Env} \times \mathbf{Prg},$$

the contexts for programs $\mathcal{P}$ and our reduction relation $\rightarrow$ for expressions, it is possible to state the rewriting rules $\rightarrow_d$ for whole programs, as shown in Figure 2.11.

The rules [**defvar**] and [**defun**] satisfy top-level definitions. The [**defvar**] rule inserts the new global binding in the variable environment ρ. The condition $x \notin dom(\rho)$ guarantees that there is only one global variable for every name. The [**defun**] rules does the equivalent for procedures — of course this time in respect to the function environment. Note that in the case of function definitions, a redefinition of the same global binding is allowed, mimicking in both cases the behavior of Emacs Lisp. The rule [**exp**] allows the use of all the reductions for expressions at the places defined using

$$
\begin{array}{ll}
[\mathbf{defvar}] & \rho, \varphi, \texttt{(defvar}\ x\ v_0^{l_0}\texttt{)}\ p \rightarrow_d \rho[x \mapsto v_0], \varphi, p \quad \text{if } x \notin dom(\rho) \\
[\mathbf{defun}] & \rho, \varphi, \texttt{(defun}\ x_0\ \texttt{(}x_1\texttt{)}\ e\texttt{)}\ p \rightarrow_d \rho, \varphi[x_0 \mapsto \texttt{(proc (}x_1\texttt{)}\ e\texttt{)}], p \\
[\mathbf{exp}] & \dfrac{\rho, \varphi, e \rightarrow \rho', \varphi', e'}{\rho, \varphi, \mathcal{P}[e] \rightarrow_d \rho', \varphi', \mathcal{P}[e']}
\end{array}
$$

Figure 2.11: Reductions for Programs.

the contexts **ExpContext**. Again $\rightarrow_d{}^*$ is the reflexive, transitive closure of $\rightarrow_d$.

2.2.7 The Evaluation Function of Programs

To determine the overall answer, the reduction relation $\rightarrow_d$ continuously rewrites the program until it gets a final answer. What is considered a satisfying answer for the result of an evaluation process? If the result of **Ans** is either a value or an exception, then the conditions for an *answer* have been satisfied.

$$a \in \mathbf{Ans} = \mathbf{Val} \cup \mathbf{Exc}.$$

It is easily shown that such an evaluation process is always of *deterministic* nature. For every program configuration there is *at most* one matching reduction rule that leads to the next configuration. This is clearly a fact following from the definition of evaluation contexts and rewriting rules. However, programs can also have an infinite reduction sequence — henceforth called *looping* — or a reduction sequence that eventually gets *stuck* at a configuration with no possible further reduction rule that matches. Taking this into account, the partial function *evaluation function eval* is defined:

$$\mathbf{eval} : \mathbf{Prg} \dashrightarrow \mathbf{Ans}$$

$$\mathbf{eval}(p) = \begin{cases} v & \mathbf{if} \quad [], [], p \rightarrow_d{}^* \rho, \varphi, v \quad \text{for some } \rho \text{ and } \varphi \\ w & \mathbf{if} \quad [], [], p \rightarrow_d{}^* \rho, \varphi, w \quad \text{for some } \rho \text{ and } \varphi \\ \mathit{undefined} & \mathbf{otherwise} \end{cases}$$

A partial function that has some undefined results seems insubstantial, but is fully adequate for the static analysis. It is easily shown by induction on the length of the reduction sequence that every program yields either exactly one answer, gets stuck or loops eternally.

Armed with tools from this chapter — the formal syntax and semantics, a formal discussion of the program analysis that correctly connects it with the programming language is possible.

Chapter 3

Dynamic Scope Analysis

Chapter 1 has identified three approaches to implement a static program analysis. The decision on the approach to take for this thesis — the static analysis of Emacs Lisp programs to gain information for a idiomatic translation — was made in favor of an constraint-based analysis. There are two main reasons for this decision: Emacs Lisp is a dynamically typed programming language. No static type system is known that could have been served as basis. Additionally, there has been many successful applications of constraint-based flow analyses in recent years [Shi88, Shi91, JW95, WC98, WS99].

A known concept of flow-analysis techniques is the repeated independent analysis of variables, expression, programs points, etc. depending on the context of either the dynamic or the static call chain of the last procedure calls. This idea appears under several different names — for example, *contours* [Shi88, Shi91, JW95], tokens or mementoes [NN97]. Shivers [Shi88] coined the name k-*CFA* for it where k is the length of the considered call chains.

Nielson and Nielson generalize the distinction depending on chains of procedure calls to a framework [NN97]. Here the exact circumstances of the creation of the contours are left abstract. There are instantiations for the known approaches.

Since the primary aim of the analysis is to determine under which binding a variable reference occurs, a new approach to gain information about the binding natures rather than about the procedure calls is necessary. Our analysis makes distinctions with respect to the different birthplaces of variables rather than the locations of the last procedure calls is implemented. Nevertheless, our approach fits nicely in the framework introduced by the Nielsons.

3.1 Specification of the Analysis

3.1.1 Abstract Domains

The original syntactic domains of the language need to be suitably abstracted depending on how much information the analysis should track. The structures used to collect entities of an analysis are sets. More complicated structures bearing an additional order are usually not used. Figure 3.1 shows the ones used for the purposes of the analysis.

$$
\begin{array}{rcl@{\quad}c@{\quad}l}
bp & \in & \widehat{\mathbf{BP}} & = & \mathbf{Lab} \cup \{\diamond\} \\
bpe & \in & \widehat{\mathbf{BPEnv}} & = & \mathbf{SymVar} \to \widehat{\mathbf{BP}} \\
\\
\hat{\mathsf{p}} & \in & \widehat{\mathbf{Cons}} & = & \mathbf{Lab} \times \mathbf{Lab} \times \widehat{\mathbf{BPEnv}} \\
\hat{\mathsf{v}} & \in & \widehat{\mathbf{Val}} & = & \mathcal{P}(\mathbf{SymVar} \cup \{\omega\} \cup \mathbf{Fun} \cup \widehat{\mathbf{Cons}}) \\
\hat{\mathsf{e}} & \in & \widehat{\mathbf{Exc}} & = & \mathcal{P}(\mathbf{Lab} \times \widehat{\mathbf{BPEnv}}) \\
\\
\hat{\rho} & \in & \widehat{\mathbf{Env}} & = & (\mathbf{SymVar} \times \widehat{\mathbf{BP}}) \to \widehat{\mathbf{Val}} \\
\hat{\varphi} & \in & \widehat{\mathbf{FEnv}} & = & \mathbf{SymVar} \to \widehat{\mathbf{Val}} \\
\hat{\mathsf{C}} & \in & \widehat{\mathbf{Cache}} & = & (\mathbf{Lab} \times \widehat{\mathbf{BPEnv}}) \to \widehat{\mathbf{Val}} \\
\hat{\mathsf{E}} & \in & \widehat{\mathbf{ECache}} & = & (\mathbf{Lab} \times \widehat{\mathbf{BPEnv}}) \to \widehat{\mathbf{Exc}}
\end{array}
$$

Figure 3.1: Abstract Domains.

Objects of the abstract set of *birthplaces* — $\widehat{\mathbf{BP}}$ for short — register syntactic locations of variable bindings. Two kinds of defining locations are registered: the $\diamond$ is for bindings occurring on the top-level and local bindings have the labels of their defining sub-expression as their representing feature.

One sort of abstraction over the regular variable environments are *birthplace environments* $\widehat{\mathbf{BPEnv}}$. A birthplace environment tells where the locations of the definitions for the actual variables are by mapping variables to elements of $\widehat{\mathbf{BP}}$.

$\widehat{\mathbf{Cons}}$ is one part of the abstract value domain; it is the set of all possible *abstract pairs* and contains all triples of two labels and a birthplace environment. A pair is always created from two sub-expression evaluated under certain variable bindings. The two labels are the labels of these two sub-expressions. The birthplace environment registers the abstract bindings that appeared at the time of the creation.

The set $\widehat{\mathbf{Val}}$ is the set of all possible *abstract values* $\hat{\mathsf{v}}$. One of these abstract values represents a set of runtime values. Not every runtime value is relevant to the analysis. Traced objects are all symbols — needed for variable names, all functions — needed to trace control flow, the single symbol ω representing all basic constants, and the previously introduced abstract pairs.

An *abstract exception* $\hat{\mathsf{e}}$ from the set $\widehat{\mathbf{Exc}}$ is a subset of witnesses for all possible occurrences of exceptions. Instead of coding them as in the semantics, the label for an exception and the birthplace environment of its alternative return value are registered.

An *abstract cache* $\hat{\mathsf{C}}$ of $\widehat{\mathbf{Cache}}$ is a abstract profile of all values that occur during a program run seen. Because the behavior with respect to different bindings is relevant, the occurring birthplace environments discriminate the occurring values. This results in different profiles for different evaluations under different environments.

The comments on abstract caches apply *mutis mutandis* to *abstract exception caches* of $\widehat{\mathbf{ECache}}$; they register exceptions instead of values. They must be kept separate, because objects of both categories flow through the programs differently.

An *abstract variable environment* $\hat{\rho}$ of $\widehat{\mathbf{Env}}$ is the union of all variable environments that show up during the evaluation of a program to one global entity associating abstract values to a combination of the variable name and its birthplace.

The *abstract function environments* $\hat{\varphi}$ of $\widehat{\mathbf{FEnv}}$ follow the same idea as the abstract variable environments. Here the evolution of the function environment is unified during instead of the variable environments. Since one cannot bind function variables locally — they are all defined on the top-level — there is no need to distinguish them by birthplaces.

Now all elements needed are assembled, in order to formally define the analysis. This will enable a formal reasoning about analyses and later give an algorithm that computes an analysis with certain properties. Hence a *dynamic scope analysis* is a four-tuple $(\hat{\mathsf{C}}, \hat{\mathsf{E}}, \hat{\rho}, \hat{\varphi})$.

3.1.2 The Acceptability Relation for Programs

The definition of the analysis by itself is insufficient for the purpose of this thesis. Considering the scope analysis $(\widehat{\mathbf{Cache}}_\perp, \widehat{\mathbf{ECache}}_\perp, \widehat{\mathbf{Env}}_\perp, \widehat{\mathbf{FEnv}}_\perp)$ where $\widehat{\mathbf{Cache}}_\perp$, $\widehat{\mathbf{ECache}}_\perp$ stand for mappings to only empty sets, it is immediately clear, that a definition is required, that clarifies which scope analyses are *valid* and which are not. For this purpose $\models$ is the *acceptability relation for programs*:

$$\models \subseteq \widehat{\mathbf{Cache}} \times \widehat{\mathbf{ECache}} \times \widehat{\mathbf{Env}} \times \widehat{\mathbf{FEnv}} \times \widehat{\mathbf{BPEnv}} \times \mathbf{Prg}.$$

It expresses the validity of scope analyses $(\hat{\mathsf{C}}, \hat{\mathsf{E}}, \hat{\rho}, \hat{\varphi})$ regarding a program p with respect to a current birthplace environment bpe. From now on, the notation is

$$(\hat{\mathsf{C}}, \hat{\mathsf{E}}, \hat{\rho}, \hat{\varphi}) \models_{bpe} p.$$

Figures 3.2 to 3.11 shall present the different cases of the relation depending on the syntactic form of p.

Value Expressions

$$
\begin{array}{lll}
[\mathbf{c}] & (\hat{\mathsf{C}}, \hat{\mathsf{E}}, \hat{\rho}, \hat{\varphi}) \models_{bpe} c^l \quad \textbf{iff} \quad \omega \in \hat{\mathsf{C}}(l, bpe) \\
[\mathbf{quote}] & (\hat{\mathsf{C}}, \hat{\mathsf{E}}, \hat{\rho}, \hat{\varphi}) \models_{bpe} \texttt{(quote } s\texttt{)}^l \quad \textbf{iff} \quad s \in \hat{\mathsf{C}}(l, bpe) \\
[\mathbf{lam}] & (\hat{\mathsf{C}}, \hat{\mathsf{E}}, \hat{\rho}, \hat{\varphi}) \models_{bpe} \texttt{(lambda (}x\texttt{) } e_0\texttt{)}^l \quad \textbf{iff} \quad \texttt{(proc (}x\texttt{) } e_0\texttt{)} \in \hat{\mathsf{C}}(l, bpe)
\end{array}
$$

Figure 3.2: Acceptability Relation — Value Expressions.

The clauses [**c**], [**quote**], and [**lam**] express the conditions for those expression that immediately evaluate to corresponding values. The clauses determine that their abstract counterpart be registered in the abstract cache under the program point l and the current birthplace environment bpe. Note that the sole occurrence of a λ-term does not require that the analysis is also valid for its body.

Expressions

$$
\begin{array}{lll}
[\mathbf{var}] & (\hat{\mathsf{C}}, \hat{\mathsf{E}}, \hat{\rho}, \hat{\varphi}) \models_{bpe} x^l \quad \textbf{iff} \quad \hat{\rho}(x, bpe(x)) \subseteq \hat{\mathsf{C}}(l, bpe) \\
[\mathbf{setq}] & (\hat{\mathsf{C}}, \hat{\mathsf{E}}, \hat{\rho}, \hat{\varphi}) \models_{bpe} \texttt{(setq } x \; t_0^{l_0}\texttt{)}^l \quad \textbf{iff} & (\hat{\mathsf{C}}, \hat{\mathsf{E}}, \hat{\rho}, \hat{\varphi}) \models_{bpe} t_0^{l_0} \wedge \\
& & \hat{\mathsf{C}}(l_0, bpe) \subseteq \hat{\rho}(x, bpe(x)) \wedge \\
& & \hat{\mathsf{C}}(l_0, bpe) \subseteq \hat{\mathsf{C}}(l, bpe) \wedge \\
& & \hat{\mathsf{E}}(l_0, bpe) \subseteq \hat{\mathsf{E}}(l, bpe)
\end{array}
$$

Figure 3.3: Acceptability Relation — Expressions (Part 1).

The occurrences of variable references and mutations put their own sets of constraints on the validity of a scope analysis. The rule [**var**] expresses a connection between the abstract value environment and the abstract cache if the expression is a variable reference. The abstract value for the variable x and its current birthplace $bpe(x)$ must be subset of the abstract value that linked it to its label and birthplace environment in the abstract cache.

If there is a `setq` in the analyzed program, the [**setq**] clause assures that the sub-expression which computes the new value for the variable remains valid under the analysis, too. Further, a valid analysis allows values that result from the sub-expression t_0 to be possible values for the variable x under the current bindings bpe and also for the whole expression. The last sub-clause forces the analysis to propagate all exceptions which come up during evaluation t_0 by $\hat{\mathsf{E}}(l_0, bpe)$ further to $\hat{\mathsf{E}}(l, bpe)$ for the entire sub-expression.

$$
\begin{array}{llll}
[\mathbf{app}] & (\hat{\mathsf{C}}, \hat{\mathsf{E}}, \hat{\rho}, \hat{\varphi}) \models_{bpe} (x_0^{l_0} \ \ t_1^{l_1})^l & \mathbf{iff} & \hat{\varphi}(x_0) \subseteq \hat{\mathsf{C}}(l_0, bpe) \land \\
& & & (\forall (\texttt{proc} \ (x_1) \ t_0^{l_b}) \in \hat{\mathsf{C}}(l_0, bpe). \\
& & & \quad (\hat{\mathsf{C}}, \hat{\mathsf{E}}, \hat{\rho}, \hat{\varphi}) \models_{bpe} t_1^{l_1} \land \\
& & & \quad (\hat{\mathsf{C}}, \hat{\mathsf{E}}, \hat{\rho}, \hat{\varphi}) \models_{bpe_0} t_0^{l_b} \land \\
& & & \quad \hat{\mathsf{C}}(l_1, bpe) \subseteq \hat{\rho}(x_1, l_b) \land \\
& & & \quad \hat{\mathsf{E}}(l_1, bpe) \subseteq \hat{\mathsf{E}}(l, bpe) \land \\
& & & \quad \hat{\mathsf{C}}(l_b, bpe_0) \subseteq \hat{\mathsf{C}}(l, bpe) \land \\
& & & \quad \hat{\mathsf{E}}(l_b, bpe_0) \subseteq \hat{\mathsf{E}}(l, bpe) \\
& & & \quad \mathbf{where} \quad bpe_0 = bpe[x_1 \mapsto l_b]) \land \\
& & & (\forall s \in \hat{\mathsf{C}}(l_0, bpe). \\
& & & \quad (\hat{\mathsf{C}}, \hat{\mathsf{E}}, \hat{\rho}, \hat{\varphi}) \models_{bpe} t_1^{l_1} \land \\
& & & \quad (\hat{\mathsf{C}}, \hat{\mathsf{E}}, \hat{\rho}, \hat{\varphi}) \models_{bpe} (s^{l_0} \ \ t_1^{l_1})^{l_s} \land \\
& & & \quad \hat{\mathsf{C}}(l_s, bpe) \subseteq \hat{\mathsf{C}}(l, bpe) \land \\
& & & \quad \hat{\mathsf{E}}(l_s, bpe) \subseteq \hat{\mathsf{E}}(l, bpe))
\end{array}
$$

Figure 3.4: Acceptability Relation — Expressions (Part 2).

The clause [**app**] shows the required constraints for procedure calls. Its first condition, $\hat{\varphi}(x_0) \subseteq \hat{\mathsf{C}}(l_0, bpe)$, guarantees that all possible values associated with x_0 in the abstract function environment are possible objects for the program point with label l_0 under the current birthplace environment bpe. Depending on the different types of the objects that possibly turn up at the operator position, a valid dynamic scope analysis enforces different further conditions.

Each procedure $(\texttt{proc} \ (x_1) \ t_0^{l_b})$ at the operator position demands that the analysis is also valid under the same birthplace environment for the operand t_1. Because the evaluation of the body of the procedure t_0 happens under an expanded environment (the parameter of the procedure creates a new variable binding), the analysis must also hold for t_0 but under a new birthplace environment bpe_0. The new bpe_0 reflects the changed binding behavior after having registered the variable x_1 under the new birthplace l_b. The label of the body of a procedure or of a `let` expression always serves as the representation of the objects introducing the new binding, since this label persists continually during the lifetime of the procedure or of the `let` expression. Further constraints put on the analysis induce the linkage of the abstract values of the argument with the formal parameter x_1 by $\hat{\mathsf{C}}(l_1, bpe) \subseteq \hat{\rho}(x_1, l_b)$ and the linkage of the possible results of the body with the whole expression by $\hat{\mathsf{C}}(l_b, bpe_0) \subseteq \hat{\mathsf{C}}(l, bpe)$. In addition, the two constraints $\hat{\mathsf{E}}(l_1, bpe) \subseteq \hat{\mathsf{E}}(l, bpe)$ and $\hat{\mathsf{E}}(l_b, bpe_0) \subseteq \hat{\mathsf{E}}(l, bpe)$ ensure that each exception, during either the evaluation of t_1 or t_0, propagates further and escapes out of the entire expression.

Because function indirection exists in Emacs Lisp, a symbol at the operator position has an effect. In such cases, the procedure is called again with the new symbol in the operator position. Therefore the analysis must also hold for the modified call, whose label is modified — the original one indexed by the symbol s. The retrieved abstract values and exceptions of the modified call must also count for the original call.

$$
\begin{array}{lll}
[\textbf{let}] & (\hat{\mathsf{C}}, \hat{\mathsf{E}}, \hat{\rho}, \hat{\varphi}) \models_{bpe} (\texttt{let}\ x\ t_1^{l_1}\ t_2^{l_2})^l & \textbf{iff} \quad (\hat{\mathsf{C}}, \hat{\mathsf{E}}, \hat{\rho}, \hat{\varphi}) \models_{bpe} t_1^{l_1} \land \\
 & & \qquad (\hat{\mathsf{C}}, \hat{\mathsf{E}}, \hat{\rho}, \hat{\varphi}) \models_{bpe_0} t_2^{l_2} \land \\
 & & \qquad \hat{\mathsf{C}}(l_1, bpe) \subseteq \hat{\rho}(x, l_2) \land \\
 & & \qquad \hat{\mathsf{E}}(l_1, bpe) \subseteq \hat{\mathsf{E}}(l, bpe) \land \\
 & & \qquad \hat{\mathsf{C}}(l_2, bpe_0) \subseteq \hat{\mathsf{C}}(l, bpe) \land \\
 & & \qquad \hat{\mathsf{E}}(l_2, bpe_0) \subseteq \hat{\mathsf{E}}(l, bpe) \\
 & & \qquad \textbf{where} \quad bpe_0 = bpe[x \mapsto l_2]
\end{array}
$$

Figure 3.5: Acceptability Relation — Expression (Part 3).

As clause [**let**] shows, a `let` expression is the second construct which introduces new local bindings. For this reason, the rule brings in the birthplace environment bpe_0, modifying the current bpe, by associating the variable x with the birthplace l_2 that stands for the body of the `let` expression. For the analysis to become valid, it must first apply to the expression t_1 transforming in value of the new binding under the current bpe, as well as for the body of `let` expression t_2, under the extended bpe_0. The remaining four sub-clauses express how the values and exceptions flow between the components of the analysis to make it valid. Identical to procedure calls, the abstract values for t_1 are possible objects for the variable x under the new birthplace l_2 — as stated by $\hat{\mathsf{C}}(l_1, bpe) \subseteq \hat{\rho}(x, l_2)$ and $\hat{\mathsf{C}}(l_2, bpe_0) \subseteq \hat{\mathsf{C}}(l, bpe)$. The possible abstract results for the body belong also to the whole expression. The two missing constraints concerning the exceptions state, the every exception occurring in an state that exceptions of sub-expressions are further propagated.

The clause [**if**] reflects the conditional behavior of the `if` expression for a valid analysis. Thus analogous to the evaluation process, the conditional sub-term t_0 must always be tested. Exceptions that occur in this analysis are also possible exceptions for the whole expression. In contrast, the two sub-terms t_1 and t_2 contribute to a valid analysis with their abstract values and exceptions depending on the abstract values that the given analysis collects for the conditional t_0 under the current bindings in $\hat{\mathsf{C}}(l_0, bpe)$. If there is an abstract value in it not equal to `nil`, the sub-term t_1 must contribute to the analysis, and if the symbol `nil` occurs, then the other term t_2 must contribute to it.

$$
\begin{array}{llll}
[\textbf{if}] & (\hat{\mathsf{C}}, \hat{\mathsf{E}}, \hat{\rho}, \hat{\varphi}) \models_{bpe} (\texttt{if } t_0^{l_0}\ t_1^{l_1}\ t_2^{l_2})^l & \textbf{iff} & (\hat{\mathsf{C}}, \hat{\mathsf{E}}, \hat{\rho}, \hat{\varphi}) \models_{bpe} t_0^{l_0} \land \\
& & & \hat{\mathsf{E}}(l_0, bpe) \subseteq \hat{\mathsf{E}}(l, bpe) \land \\
& & & (\exists \hat{\mathsf{v}} \in \hat{\mathsf{C}}(l_0, bpe). \hat{\mathsf{v}} \neq \texttt{nil} \Rightarrow \\
& & & \quad (\hat{\mathsf{C}}, \hat{\mathsf{E}}, \hat{\rho}, \hat{\varphi}) \models_{bpe} t_1^{l_1} \land \\
& & & \quad \hat{\mathsf{C}}(l_1, bpe) \subseteq \hat{\mathsf{C}}(l, bpe) \land \\
& & & \quad \hat{\mathsf{E}}(l_1, bpe) \subseteq \hat{\mathsf{E}}(l, bpe)) \land \\
& & & (\texttt{nil} \in \hat{\mathsf{C}}(l_0, bpe) \Rightarrow \\
& & & \quad (\hat{\mathsf{C}}, \hat{\mathsf{E}}, \hat{\rho}, \hat{\varphi}) \models_{bpe} t_2^{l_2} \land \\
& & & \quad \hat{\mathsf{C}}(l_2, bpe) \subseteq \hat{\mathsf{C}}(l, bpe) \land \\
& & & \quad \hat{\mathsf{E}}(l_2, bpe) \subseteq \hat{\mathsf{E}}(l, bpe)) \\
[\textbf{catch}] & (\hat{\mathsf{C}}, \hat{\mathsf{E}}, \hat{\rho}, \hat{\varphi}) \models_{bpe} (\texttt{catch } t_0^{l_0}\ t_1^{l_1})^l & \textbf{iff} & (\hat{\mathsf{C}}, \hat{\mathsf{E}}, \hat{\rho}, \hat{\varphi}) \models_{bpe} t_0^{l_0} \land \\
& & & \hat{\mathsf{E}}(l_0, bpe) \subseteq \hat{\mathsf{E}}(l, bpe) \land \\
& & & (\hat{\mathsf{C}}, \hat{\mathsf{E}}, \hat{\rho}, \hat{\varphi}) \models_{bpe} t_1^{l_1} \land \\
& & & \hat{\mathsf{C}}(l_1, bpe) \subseteq \hat{\mathsf{C}}(l, bpe) \land \\
& & & (\forall (l_2, bpe') \in \hat{\mathsf{E}}(l_1, bpe). \\
& & & \quad \hat{\mathsf{C}}(l_2, bpe') \subseteq \hat{\mathsf{C}}(l, bpe)) \land \\
& & & \hat{\mathsf{E}}(l_1, bpe) \subseteq \hat{\mathsf{E}}(l, bpe)
\end{array}
$$

Figure 3.6: Acceptability Relation — Expressions (Part 4).

The other control structure `catch` provides some constraints to a valid scope analysis through the [**catch**] clause. The analysis must be "recursively" valid for both sub-terms t_0 — the one standing for the tags of those exceptions to handle, and t_1, the expression contributing the result in the normal case. If during the evaluation of the tag t_0 an exception occurs, it must hold for the whole expression, as $\hat{\mathsf{E}}(l_0, bpe) \subseteq \hat{\mathsf{E}}(l, bpe)$ requires. All "normal" abstract values for t_1 are possible results for the entire `catch` expression, as shown by the constraint $\hat{\mathsf{C}}(l_1, bpe) \subseteq \hat{\mathsf{C}}(l, bpe)$. The abnormal case is when an exception occurs during the evaluation of t_1. A concession is made to keep possible formulations of the analysis simpler by not distinguishing different exceptions by their tags abstractly. Instead, it is possible that an exception is handled, so that its alternative return value is a possible abstract return value for the `catch` expression, as expressed by $(\forall (l_2, bpe') \in \hat{\mathsf{E}}(l_1, bpe). \hat{\mathsf{C}}(l_2, bpe') \subseteq \hat{\mathsf{C}}(l, bpe))$. $\hat{\mathsf{E}}(l_1, bpe) \subseteq \hat{\mathsf{E}}(l, bpe)$ expresses that every exception can further escape to the surrounding block.

Primitives

Primitives are built-in procedures. Each one has it its own semantical properties and therefore its own flow behavior.

The clause [**funcall**] describing the properties of the applications of the primitive `funcall` is for regular procedure calls. The analysis does not

$$
\begin{array}{ll}
\textbf{[funcall]} & (\hat{\mathsf{C}}, \hat{\mathsf{E}}, \hat{\rho}, \hat{\varphi}) \models_{bpe} \texttt{(funcall } t_0^{l_0} \; t_1^{l_1}\texttt{)}^l \quad \textbf{iff} \\
& \quad (\hat{\mathsf{C}}, \hat{\mathsf{E}}, \hat{\rho}, \hat{\varphi}) \models_{bpe} t_0^{l_0} \wedge \\
& \quad \hat{\mathsf{E}}(l_0, bpe) \subseteq \hat{\mathsf{E}}(l, bpe) \wedge \\
& \quad (\hat{\mathsf{C}}, \hat{\mathsf{E}}, \hat{\rho}, \hat{\varphi}) \models_{bpe} t_1^{l_1} \wedge \\
& \quad \hat{\mathsf{E}}(l_1, bpe) \subseteq \hat{\mathsf{E}}(l, bpe) \wedge \\
& \quad (\forall \texttt{(proc (}x_1\texttt{) } t_b^{l_b}\texttt{)} \in \hat{\mathsf{C}}(l_0, bpe). \\
& \qquad (\hat{\mathsf{C}}, \hat{\mathsf{E}}, \hat{\rho}, \hat{\varphi}) \models_{bpe_0} t_b^{l_b} \wedge \\
& \qquad \hat{\mathsf{C}}(l_1, bpe) \subseteq \hat{\rho}(x_1, l_b) \wedge \\
& \qquad \hat{\mathsf{C}}(l_b, bpe_0) \subseteq \hat{\mathsf{C}}(l, bpe) \wedge \\
& \qquad \hat{\mathsf{E}}(l_b, bpe_0) \subseteq \hat{\mathsf{E}}(l, bpe) \\
& \qquad \textbf{where} \quad bpe_0 = bpe[x_1 \mapsto l_b]) \wedge \\
& \quad (\forall s \in \hat{\mathsf{C}}(l_0, bpe). \forall v \in \mathbf{Val}. \\
& \qquad v \, \mathcal{V} \, (\hat{\varphi}(s), \hat{\mathsf{C}}) \Rightarrow \\
& \qquad \quad (\hat{\mathsf{C}}, \hat{\mathsf{E}}, \hat{\rho}, \hat{\varphi}) \models_{bpe} \texttt{(funcall } v^{t_0} \; t_1^{l_1}\texttt{)}^{l_s} \wedge \\
& \qquad \quad \hat{\mathsf{C}}(l_s, bpe) \subseteq \hat{\mathsf{C}}(l, bpe) \wedge \\
& \qquad \quad \hat{\mathsf{E}}(l_s, bpe) \subseteq \hat{\mathsf{E}}(l, bpe))
\end{array}
$$

Figure 3.7: Acceptability Relation — Primitives (Part 1).

access the functional environment to obtain a procedure but uses the result of the evaluation of the term t_1. The first four conditions of the clause are typical for primitives; the analysis must also be valid for the sub-terms t_0 and t_1 of the expression, representing the arguments of the primitive. The exceptions that occur during the evaluation of the argument are also possible exceptions of the entire expression. Depending on what kind of abstract value occurs in the abstract cache for the first argument, further constraints must hold for $(\hat{\mathsf{C}}, \hat{\mathsf{E}}, \hat{\rho}, \hat{\varphi})$. If the first argument is a regular procedure, the analysis must hold for the body of the procedure t_b under the extended birthplace environment bpe_0 in the same manner as in the first part of the clause [**app**]. If a symbol instead of a procedure is found for t_0, once again "function indirection" step can take place. To handle this right, the analysis must be valid for all other `funcall`s that modify the original one by possible candidates v, found in the function environment as operator parameter instead of t_0. To get all values v that possibly occur in the function environment, the relation $\mathcal{V}$ (defined in the following subsection) filters all values **Val** that are member of $\hat{\varphi}(s)$. As in the case for [**app**], the modified `funcall`s must be valid under a modified label — the original label l indexed by the new symbol s — and every thus resulting value and exception must contribute to the overall set of values and exceptions for the expression.

Except for the obligatory conditions of the sub-expressions and occurring exceptions, the [**cons**] rule handling `cons` expressions is rather straightfor-

$$
\begin{array}{llll}
[\textbf{cons}] & (\hat{\mathsf{C}},\hat{\mathsf{E}},\hat{\rho},\hat{\varphi}) \models_{bpe} (\texttt{cons}\ t_1^{l_1}\ t_2^{l_2})^l & \textbf{iff} & (\hat{\mathsf{C}},\hat{\mathsf{E}},\hat{\rho},\hat{\varphi}) \models_{bpe} t_1^{l_1} \land \\
 & & & (\hat{\mathsf{C}},\hat{\mathsf{E}},\hat{\rho},\hat{\varphi}) \models_{bpe} t_2^{l_2} \land \\
 & & & (l_1, l_2, bpe) \in \hat{\mathsf{C}}(l, bpe) \land \\
 & & & \hat{\mathsf{E}}(l_1, bpe) \subseteq \hat{\mathsf{E}}(l, bpe) \land \\
 & & & \hat{\mathsf{E}}(l_2, bpe) \subseteq \hat{\mathsf{E}}(l, bpe) \\
 & & & \\
[\textbf{car}] & (\hat{\mathsf{C}},\hat{\mathsf{E}},\hat{\rho},\hat{\varphi}) \models_{bpe} (\texttt{car}\ t_0^{l_0})^l & \textbf{iff} & (\hat{\mathsf{C}},\hat{\mathsf{E}},\hat{\rho},\hat{\varphi}) \models_{bpe} t_0^{l_0} \land \\
 & & & (\forall (l_1, l_2, bpe') \in \hat{\mathsf{C}}(l_0, bpe). \\
 & & & \quad \hat{\mathsf{C}}(l_1, bpe') \subseteq \hat{\mathsf{C}}(l, bpe)) \land \\
 & & & (\texttt{nil} \in \hat{\mathsf{C}}(l_0, bpe) \Rightarrow \\
 & & & \quad \texttt{nil} \in \hat{\mathsf{C}}(l, bpe)) \land \\
 & & & \hat{\mathsf{E}}(l_0, bpe) \subseteq \hat{\mathsf{E}}(l, bpe) \\
 & & & \\
[\textbf{throw}] & (\hat{\mathsf{C}},\hat{\mathsf{E}},\hat{\rho},\hat{\varphi}) \models_{bpe} (\texttt{throw}\ t_1^{l_1}\ t_2^{l_2})^l & \textbf{iff} & (\hat{\mathsf{C}},\hat{\mathsf{E}},\hat{\rho},\hat{\varphi}) \models_{bpe} t_1^{l_1} \land \\
 & & & (\hat{\mathsf{C}},\hat{\mathsf{E}},\hat{\rho},\hat{\varphi}) \models_{bpe} t_2^{l_2} \land \\
 & & & \hat{\mathsf{E}}(l_1, bpe) \subseteq \hat{\mathsf{E}}(l, bpe) \land \\
 & & & \hat{\mathsf{E}}(l_2, bpe) \subseteq \hat{\mathsf{E}}(l, bpe) \land \\
 & & & (l_2, bpe) \in \hat{\mathsf{E}}(l, bpe)
\end{array}
$$

Figure 3.8: Acceptability Relation — Primitives (Part 2).

ward. An abstract pair created from abstract values with the labels l_1 and l_2 under bpe must be a possible abstract value for the entire expression.

The clause [**car**] is also quite simple. Here a valid scope analysis must further propagate all abstract values being first component of an abstract pair to let them become possible values of the `car` expression.

Finally, the [**throw**] clause treating its homonymic primitive assures that — in addition to the standard primitive conditions — the valid analysis registers with $(l_2, bpe) \in \hat{\mathsf{E}}(l, bpe)$, an exception for the label l under bpe carrying the alternative return value created by the second sub-expression $t_2^{l_2}$.

Definitions

Of course the goal is no only to analyze single expressions, but entire programs. Consequently, definitions need constraints as well to extend the notion of valid scope analyses to whole programs.

The clause [**defvar**] handles those programs beginning with a variable definition. The definitions determine the initial value of the new variable by evaluating the term t_0. Therefore a valid analysis must consider this term — the first sub-clause $(\hat{\mathsf{C}},\hat{\mathsf{E}},\hat{\rho},\hat{\varphi}) \models_{bpe} t_0^{l_0}$; it must state that the possibly obtained abstract values is associated with the variable x under the current binding $bpe(x)$, and it must also assure that possibly occurring exceptions

$$\begin{array}{llll}
[\textbf{defvar}] & (\hat{\mathsf{C}}, \hat{\mathsf{E}}, \hat{\rho}, \hat{\varphi}) \models_{bpe} \texttt{(defvar } x \ t_0^{l_0}\texttt{)} \ p & \textbf{iff} & (\hat{\mathsf{C}}, \hat{\mathsf{E}}, \hat{\rho}, \hat{\varphi}) \models_{bpe} t_0^{l_0} \wedge \\
& & & \hat{\mathsf{C}}(l_0, bpe) \subseteq \hat{\rho}(x, bpe(x)) \wedge \\
& & & \hat{\mathsf{E}}(l_0, bpe) \subseteq \hat{\mathsf{E}}(\diamond, bpe) \wedge \\
& & & (\hat{\mathsf{C}}, \hat{\mathsf{E}}, \hat{\rho}, \hat{\varphi}) \models_{bpe} p
\end{array}$$

$$\begin{array}{lll}
[\textbf{defun}] & (\hat{\mathsf{C}}, \hat{\mathsf{E}}, \hat{\rho}, \hat{\varphi}) \models_{bpe} \texttt{(defun } x_0 \ \texttt{(}x_1\texttt{)} \ e\texttt{)} \ p & \textbf{iff} \\
& \quad \texttt{(proc (}x_1\texttt{)} \ e\texttt{)} \in \hat{\varphi}(x_0) \wedge & \\
& \quad (\hat{\mathsf{C}}, \hat{\mathsf{E}}, \hat{\rho}, \hat{\varphi}) \models_{bpe} p &
\end{array}$$

Figure 3.9: Acceptability Relation — Definitions.

flow through to the top-level symbolized by the special label $\diamond$. Naturally, a valid analysis must take into account the rest of the program named p, too.

The formulation of the [**defun**] clause treating the definition of global procedures is even simpler. Since there is no further evaluation of the procedure involved, the procedure becomes globally bound to its function name. The valid analysis registers the procedure in the abstract function environment. Just as above, the rest of the program a must be valid, too.

Values and Exceptions

So far, the definition of the relation $\models$ for all possible expressions and programs checks whether a certain analysis for a program is valid or not. However, the ultimate goal is to show that valid analyses agree with the semantics — that is, that they are semantically correct, and to do that the definition so far is insufficient. During an evaluation, expressions, values, exceptions and intermediate expressions take place. Therefore, the last part of this section contains the definition of $\models$ for those syntactic categories.

The rules [**const**], [**sym**], and [**proc**] are identical to their equivalents [**c**], [**quote**], and [**lam**] of the corresponding value expressions, because their semantics are identical.

The [**pair**] clause is a simpler version of the clause [**cons**]. The difference is — since a pair already carries two values in it — that even though the evaluation of sub-expressions drops out, it is unknown under which prior birthplace environment the evaluation took place. The requirements are that just a bpe_0 exists.

The [**exc**] clause adapts the [**throw**] rule for `throw` expressions to exceptions by following the idea of the previous rule.

Intermediate Expressions

The last clause takes care of the immediate expression `bind`.

$$
\begin{array}{lll}
[\textbf{const}] & (\hat{\mathsf{C}}, \hat{\mathsf{E}}, \hat{\rho}, \hat{\varphi}) \models_{bpe} (\texttt{const}\ c)^l \ \ \textbf{iff} & \omega \in \hat{\mathsf{C}}(l, bpe) \\[1ex]
[\textbf{sym}] & (\hat{\mathsf{C}}, \hat{\mathsf{E}}, \hat{\rho}, \hat{\varphi}) \models_{bpe} (\texttt{sym}\ s)^l \ \ \textbf{iff} & s \in \hat{\mathsf{C}}(l, bpe) \\[1ex]
[\textbf{proc}] & (\hat{\mathsf{C}}, \hat{\mathsf{E}}, \hat{\rho}, \hat{\varphi}) \models_{bpe} (\texttt{proc}\ (x)\ e_0)^l \ \ \textbf{iff} & (\texttt{proc}\ (x)\ e_0) \in \hat{\mathsf{C}}(l, bpe) \\[1ex]
[\textbf{pair}] & (\hat{\mathsf{C}}, \hat{\mathsf{E}}, \hat{\rho}, \hat{\varphi}) \models_{bpe} (\texttt{pair}\ v_1^{l_1}\ v_2^{l_2})^l \ \ \textbf{iff} & \exists bpe_0.(\hat{\mathsf{C}}, \hat{\mathsf{E}}, \hat{\rho}, \hat{\varphi}) \models_{bpe_0} v_1^{l_1} \wedge \\
& & (\hat{\mathsf{C}}, \hat{\mathsf{E}}, \hat{\rho}, \hat{\varphi}) \models_{bpe_0} v_2^{l_2} \wedge \\
& & (l_1, l_2, bpe_0) \in \hat{\mathsf{C}}(l, bpe) \\[1ex]
[\textbf{exc}] & (\hat{\mathsf{C}}, \hat{\mathsf{E}}, \hat{\rho}, \hat{\varphi}) \models_{bpe} (\texttt{exc}\ v_1^{l_1}\ v_2^{l_2})^l \ \ \textbf{iff} & \exists bpe_0.(\hat{\mathsf{C}}, \hat{\mathsf{E}}, \hat{\rho}, \hat{\varphi}) \models_{bpe_0} v_1^{l_1} \wedge \\
& & (\hat{\mathsf{C}}, \hat{\mathsf{E}}, \hat{\rho}, \hat{\varphi}) \models_{bpe_0} v_2^{l_2} \wedge \\
& & (l_2, bpe_0) \in \hat{\mathsf{E}}(l, bpe)
\end{array}
$$

Figure 3.10: Acceptability Relation — Values and Exceptions.

The [**bind**] clause handles the `bind`-construct that equips a term t_2 with an additional local binding of the variable x to the value v_1. It shows that the valid analysis must also examine the term correctly but must take into account an extended birthplace environment bpe_0 because of the introduction of the expression of the new one. Whatever is the result of the body is also a possible result for the whole expression. The supplementary constraint $v_1\, \mathcal{V}\, (\hat{\rho}(x, l_2), \hat{\mathsf{C}})$ reflects that the actual new binding also has to show up in the abstract variable environment under the the relevant birthplace l_2.

$$
\begin{array}{lll}
[\textbf{bind}] & (\hat{\mathsf{C}}, \hat{\mathsf{E}}, \hat{\rho}, \hat{\varphi}) \models_{bpe} (\texttt{bind}\ x\ v_1\ t_2^{l_2})^l \ \ \textbf{iff} & (\hat{\mathsf{C}}, \hat{\mathsf{E}}, \hat{\rho}, \hat{\varphi}) \models_{bpe_0} t_2^{l_2} \wedge \\
& & \hat{\mathsf{C}}(l_2, bpe_0) \subseteq \hat{\mathsf{C}}(l, bpe) \wedge \\
& & \hat{\mathsf{E}}(l_2, bpe_0) \subseteq \hat{\mathsf{E}}(l, bpe) \wedge \\
& & v_1\, \mathcal{V}\, (\hat{\rho}(x, l_2), \hat{\mathsf{C}}) \\
& & \textbf{where}\quad bpe_0 = bpe[x \mapsto l_2]
\end{array}
$$

Figure 3.11: Acceptability Relation — Intermediate Expressions.

3.1.3 The Approximation Relation $\mathcal{V}$

Intuitively, the relation $\models$ states if a dynamic scope analysis $(\hat{\mathsf{C}}, \hat{\mathsf{E}}, \hat{\rho}, \hat{\varphi})$ reflects the whole evaluation process of a program a in an abstract view. The formulation of $\models$ uses the *approximation relation* $\mathcal{V}$ that express the correct approximation of the syntactical category of values **Val** with their abstract equivalents. Figure 3.12 shows its formal definition.

$$\mathcal{V} \subseteq \mathbf{Val} \times \widehat{\mathbf{Val}} \times \widehat{\mathbf{Cache}}$$

$$\begin{array}{lll} v\,\mathcal{V}\,(\hat{\mathsf{v}}, \hat{\mathsf{C}}) & \textbf{iff} & \forall c\,\forall s\,\forall f\,\forall v_1\,\forall v_2.((v = c \Rightarrow \omega \in \hat{\mathsf{v}}) \wedge \\ & & \quad (v = s \Rightarrow s \in \hat{\mathsf{v}}) \wedge \\ & & \quad (v = f \Rightarrow f \in \hat{\mathsf{v}}) \wedge \\ & & \quad (v = (\texttt{pair}\ v_1^{l_1}\ v_2^{l_2}) \Rightarrow \\ & & \qquad \exists bpe.(l_1, l_2, bpe) \in \hat{\mathsf{v}} \wedge \\ & & \qquad\quad v_1\,\mathcal{V}\,(\hat{\mathsf{C}}(l_1, bpe), \hat{\mathsf{C}}) \wedge \\ & & \qquad\quad v_2\,\mathcal{V}\,(\hat{\mathsf{C}}(l_2, bpe), \hat{\mathsf{C}}))) \end{array}$$

Figure 3.12: Approximation Relation for Values.

The relation $\mathcal{V}$ serves us as to express when a certain value v is correctly represented by a set of abstract values $\hat{\mathsf{v}}$. The only case where this is a little bit more complicated is the treatment of pairs. Because a pair has the labels of its components' creation points and a birthplace environment as abstract representation also ask its sub-components v_1 and v_2 must have a correct approximation for some bpe in the abstract value cache. This is the reason why $\mathcal{V}$ also needs to incorporate a value cache $\hat{\mathsf{C}}$. The relation is well-defined because its definition is recursive with respect to the structure of the values v.

3.1.4 The Well-Definedness of $\models$

It is not at all clear, that the acceptability relation $\models$ from Subsection 3.1.2 is well-defined — that its definition is unambiguous. In fact, structural induction is not sufficient to justify the definition. A part of the definition that raises a problem is for example the [**app**] clause because its right-hand side does not consist of strictly smaller syntactic sub-terms of its the left-hand side.

On the other hand, the specifications of $\models$ can be considered as the functional

$$\begin{array}{l} \mathcal{Q} : \mathcal{P}(\widehat{\mathbf{Cache}} \times \widehat{\mathbf{ECache}} \times \widehat{\mathbf{Env}} \times \widehat{\mathbf{FEnv}} \times \widehat{\mathbf{BPEnv}} \times \mathbf{Exp}) \rightarrow \\ \qquad \mathcal{P}(\widehat{\mathbf{Cache}} \times \widehat{\mathbf{ECache}} \times \widehat{\mathbf{Env}} \times \widehat{\mathbf{FEnv}} \times \widehat{\mathbf{BPEnv}} \times \mathbf{Exp}) \end{array}$$

where

$$\begin{array}{rl}\mathcal{Q}(R_1) = & (\hat{\mathsf{C}}, \hat{\mathsf{E}}, \hat{\rho}, \hat{\varphi}, bpe, (\texttt{let}\ x\ t_1^{l_1}\ t_2^{l_2})^l) \in R_2 \textbf{ iff} \\ & \quad R_1(\hat{\mathsf{C}}, \hat{\mathsf{E}}, \hat{\rho}, \hat{\varphi}, bpe, t_1^{l_1}) \wedge \\ & \quad R_1(\hat{\mathsf{C}}, \hat{\mathsf{E}}, \hat{\rho}, \hat{\varphi}, bpe[x \mapsto l_2], t_2^{l_2}) \wedge \\ & \quad \hat{\mathsf{C}}(l_1, bpe) \subseteq \hat{\rho}(x, l_2) \wedge \\ & \quad \hat{\mathsf{E}}(l_1, bpe) \subseteq \hat{\mathsf{E}}(l, bpe) \wedge \\ & \quad \hat{\mathsf{C}}(l_2, bpe[x \mapsto l_2]) \subseteq \hat{\mathsf{C}}(l, bpe) \wedge \\ & \quad \hat{\mathsf{E}}(l_2, bpe[x \mapsto l_2]) \subseteq \hat{\mathsf{C}}(l, bpe) \\ & (\hat{\mathsf{C}}, \hat{\mathsf{E}}, \hat{\rho}, \hat{\varphi}, bpe, \dots) \in R_2 \textbf{ iff } \dots ,\end{array}$$

this other perspective leads to a specification of $\models$ using sound mathematical means.

The function $\mathcal{Q}$ is a monotone function on the complete lattice

$$(\mathcal{P}(\widehat{\mathbf{Cache}} \times \widehat{\mathbf{Env}} \times \widehat{\mathbf{Env}} \times \widehat{\mathbf{BPEnv}} \times \mathbf{Exp}), \sqsubseteq)$$

because

- $(\mathcal{P}(\widehat{\mathbf{Cache}} \times \widehat{\mathbf{Env}} \times \widehat{\mathbf{Env}} \times \widehat{\mathbf{BPEnv}} \times \mathbf{Exp}), \sqsubseteq)$ is a complete lattice with respect to the partial order $R_1 \sqsubseteq R_2$ **iff** $\forall t. t \in R_1 \Rightarrow t \in R_2$, and

- $\mathcal{Q}$ is a monotone function on this complete lattice — that is, $\forall R_1, R_2. R_1 \sqsubseteq R_2 \Rightarrow \mathcal{Q}(R_1) \sqsubseteq \mathcal{Q}(R_2)$.

By Tarski's fixed point theorem follows, that $\mathcal{Q}$ has a fixed point and even a greatest fixed point. Thus a well-defined definition of $\models$ works by *co-induction* as

$$\models := gfp(\mathcal{Q}).$$

The concept of co-induction arose from Milner and Tofte's works [MT91] of considering semantics and type systems of an extended λ-calculus with references. The Nielsons were the first ones who used co-induction as a means of specification to static analysis [NN97]. The interested reader should consult the appendix of the book [NNH99] of Nielson, Nielson and Hankin for further explanations.

3.1.5 The Acceptability Relation for Environments

Chapter 2 first introduced a formal syntax of Mini Emacs Lisp, and later, gave it a meaning by a formal semantics using reduction rules. Additionally, for such formal evaluation processes, the two kinds of environments — the variable and the function environment — added to register the global bindings of variables and functions, respectively.

If the acceptability relation for programs also should become semantically correct — that is, that a valid dynamic scope analysis for a program should stays valid over the whole evaluation process, — one must additionally be able to say, whether a dynamic scope analysis validly models those regular environments or not, since they are also part of all the information appearing during evaluation. The *acceptability relation for environments* $\models$ expresses this connection; Figure 3.13 shows its formal definitions.

$$\models \subseteq \widehat{\mathbf{Cache}} \times \widehat{\mathbf{ECache}} \times \widehat{\mathbf{Env}} \times \widehat{\mathbf{FEnv}} \times \widehat{\mathbf{BPEnv}} \times \mathbf{Env} \times \mathbf{Env}$$

$$(\hat{\mathsf{C}}, \hat{\mathsf{E}}, \hat{\rho}, \hat{\varphi}) \models_{bpe} \rho, \varphi \quad \textbf{iff} \quad \begin{array}{l} \forall x \in dom(\rho).\rho(x)\, \mathcal{V}\, (\hat{\rho}(x, bpe(x)), \hat{\mathsf{C}}) \wedge \\ \forall x \in dom(\varphi).\varphi(x)\, \mathcal{V}\, (\hat{\varphi}(x), \hat{\mathsf{C}}) \end{array}$$

Figure 3.13: Acceptability Relation for Environments.

As said already, the abstract value and function environments unify all real bindings — global as well as local ones — into two integral entities. If now an environment ρ or φ shows up at a certain point of the execution, there should also exists a formal that tells, whether the environment is correctly approximated by the right abstract values, which $\hat{\rho}$ and $\hat{\varphi}$ registered with the current birthplaces of bpe. $\models$ embodies this relation at the same time for variable environment and a function environment by demanding that all values associated with the environments are independently approximated correctly as stated by the usage of the approximation relation $\mathcal{V}$.

3.1.6 The Acceptability Relation for Configurations

While introducing the semantic reductions, the composition of the an environment and an expressions became configurations, and the composition with programs to the name became program configurations, respectively. Accordingly, the combination of the two existing acceptability relations — the one for programs and the one for environments — becomes the *acceptability relation for configurations* and the *acceptability relation for program configurations*, respectively as shown in in Figure 3.14

3.2 Semantic Correctness

The semantics developed in Chapter 2 employs evaluation contexts and rewriting rules. That means, the specification of the semantics only uses syntactical means (neglecting now the notion of environments of course): a program goes through a transition of changing configurations that include valid programs or expressions respectively until it reaches a final value, gets stuck or loops forever. By the definition of the acceptability relation in the first section of this chapter, the definition of a valid dynamic scope analysis was derived intuitively. But for now it seems not at all clear, why every valid

$$\models \subseteq \widehat{\mathbf{Cache}} \times \widehat{\mathbf{ECache}} \times \widehat{\mathbf{Env}} \times \widehat{\mathbf{FEnv}} \times \widehat{\mathbf{BPEnv}} \times \mathbf{Conf}$$

$$(\hat{\mathsf{C}}, \hat{\mathsf{E}}, \hat{\rho}, \hat{\varphi}) \models_{bpe} \rho, \varphi, e \quad \textbf{iff} \quad \begin{array}{l} (\hat{\mathsf{C}}, \hat{\mathsf{E}}, \hat{\rho}, \hat{\varphi}) \models_{bpe} \rho, \varphi \wedge \\ (\hat{\mathsf{C}}, \hat{\mathsf{E}}, \hat{\rho}, \hat{\varphi}) \models_{bpe} e \end{array}$$

$$\models \subseteq \widehat{\mathbf{Cache}} \times \widehat{\mathbf{ECache}} \times \widehat{\mathbf{Env}} \times \widehat{\mathbf{FEnv}} \times \widehat{\mathbf{BPEnv}} \times \mathbf{PConf}$$

$$(\hat{\mathsf{C}}, \hat{\mathsf{E}}, \hat{\rho}, \hat{\varphi}) \models_{bpe} \rho, \varphi, p \quad \textbf{iff} \quad \begin{array}{l} (\hat{\mathsf{C}}, \hat{\mathsf{E}}, \hat{\rho}, \hat{\varphi}) \models_{bpe} \rho, \varphi \wedge \\ (\hat{\mathsf{C}}, \hat{\mathsf{E}}, \hat{\rho}, \hat{\varphi}) \models_{bpe} p \end{array}$$

Figure 3.14: Acceptability Relation for Configurations and Program Configurations.

scope analysis should correctly reflect the abstract behavior of every imaginable program. But since every program traverses a sequence of changing programs, the correctness can can be proven by showing that every valid analysis stays valid under the whole process of evaluation.

Those notion of correctness is called a *subject reduction* result. Curry and Feys [CF58] introduced it to show the correctness of some properties called predicates in the languages of combinatory logic. Mitchell and Plotkin [MP88] used the idea to show a type correctness result for a λ-calculus like language, whereas Wright and Felleisen [WF91, WF94] adapt it to the more flexible concept of operational semantics with reduction rules and evaluation contexts. A number of flow analyses based on this idea followed — among them the work of Wright and Jagannathan [WJ98] defining *polymorphic splitting* that inspired our semantics and style of proof.

Before tackling the main correctness result — first for expressions then for whole programs — some lemmata are stated and proven. They will help at the later proofs. The following first lemma shall mainly help us simplifying those parts of the proof of the main theorem where values occur. It tells us that a dynamic scope analysis is valid for a value if and only if the value is part of the abstract cache at its label and the given *bpe*.

Lemma 1. $(\hat{\mathsf{C}}, \hat{\mathsf{E}}, \hat{\rho}, \hat{\varphi}) \models_{bpe} v^l \quad \textit{iff} \quad v\, \mathcal{V}\, (\hat{\mathsf{C}}(l, bpe), \hat{\mathsf{C}})$

Proof. The proof is done by structural induction over v.

Case $v =$ `(const` c`)`. From the assumption and the application of clause [**const**] of $\models$, the following equivalence follows:

$$(\hat{\mathsf{C}}, \hat{\mathsf{E}}, \hat{\rho}, \hat{\varphi}) \models_{bpe} (\texttt{const}\ c)^l \quad \textit{iff} \quad \omega \in \hat{\mathsf{C}}(l, bpe)$$

and the definition of $\mathcal{V}$ gives

$$\textit{iff} \quad c\, \mathcal{V}\, (\hat{\mathsf{C}}(l, bpe), \hat{\mathsf{C}})$$

Case $v =$ `(sym` s`)`. Analogous to the first case.

Case $v =$ `(proc (`x`)` e`)`. Analogous to the first case.

Case $v =$ `(proc` x e`)`. Analogous to the first case.

Case $v =$ `(pair` $v_1^{l_1}$ $v_2^{l_2}$`)`. The assumption together with the clause [**pair**] of the acceptability relation leads to

$$
\begin{array}{lll}
(\hat{\mathsf{C}}, \hat{\mathsf{E}}, \hat{\rho}, \hat{\varphi}) \models_{bpe} (\texttt{pair}\ v_1^{l_1}\ v_2^{l_2})^l & \textit{iff} & \text{there exists some } bpe_0 \text{ with} \\
& & (\hat{\mathsf{C}}, \hat{\mathsf{E}}, \hat{\rho}, \hat{\varphi}) \models_{bpe_0} v_1^{l_1} \quad (3.1) \\
& & (\hat{\mathsf{C}}, \hat{\mathsf{E}}, \hat{\rho}, \hat{\varphi}) \models_{bpe_0} v_2^{l_2} \quad (3.2) \\
& & (l_1, l_2, bpe_0) \in \hat{\mathsf{C}}(l, bpe)
\end{array}
$$

and this is — by applying the induction hypothesis to (3.1) and (3.2) — equivalent to

$$
\begin{array}{ll}
\textit{iff} & \text{there exists some } bpe_0 \text{ with} \\
& v_1\, \mathcal{V}\, (\hat{\mathsf{C}}(l_1, bpe), \hat{\mathsf{C}}) \\
& v_2\, \mathcal{V}\, (\hat{\mathsf{C}}(l_2, bpe), \hat{\mathsf{C}}) \\
& (l_1, l_2, bpe_0) \in \hat{\mathsf{C}}(l, bpe).
\end{array}
$$

Finally, by the definition of $\mathcal{V}$, the result is

$$
\textit{iff} \quad (\texttt{pair}\ v_1^{l_1}\ v_2^{l_2})^l\, \mathcal{V}\, (\hat{\mathsf{C}}(l, bpe), \hat{\mathsf{C}}).
$$

□

Another tool will help to simplify the formulation of the following lemmas. It just proves the obvious assumption, that if an abstract value $\hat{\mathsf{v}}_1$ is a correct approximation of a true value v, it is also a correct approximation of another abstract value $\hat{\mathsf{v}}_2$ which includes the former one.

Lemma 2. *If* $v\, \mathcal{V}\, (\hat{\mathsf{v}}_1, \hat{\mathsf{C}})$ *and also* $\hat{\mathsf{v}}_1 \subseteq \hat{\mathsf{v}}_2$ *then* $v\, \mathcal{V}\, (\hat{\mathsf{v}}_2, \hat{\mathsf{C}})$.

Proof. The lemma follows by structural induction over v. Each case of the proof is obtained individually by inspecting the definition of $\mathcal{V}$. □

The specification of the acceptability relation has the important property stated by the following Lemma 3: if an analysis is valid for a term t under the label t_1, and also the abstract values and exceptions are all contained under the label l_2, the analysis will in addition be valid under the second label l_2.

Lemma 3. *If* $(\hat{\mathsf{C}}, \hat{\mathsf{E}}, \hat{\rho}, \hat{\varphi}) \models_{bpe} t^{l_1}$, $\hat{\mathsf{C}}(l_1, bpe) \subseteq \hat{\mathsf{C}}(l_2, bpe)$, *and* $\hat{\mathsf{E}}(l_1, bpe) \subseteq \hat{\mathsf{E}}(l_2, bpe)$ *then also* $(\hat{\mathsf{C}}, \hat{\mathsf{E}}, \hat{\rho}, \hat{\varphi}) \models_{bpe} t^{l_2}$.

Proof. To prove this lemma, a case analysis differentiated by the grammatical rules of **Term** is sufficient. As an example, the case for `setq`-expressions follows:
By the first premise

$$(\hat{\mathsf{C}}, \hat{\mathsf{E}}, \hat{\rho}, \hat{\varphi}) \models_{bpe} \texttt{(setq } x\ t_0^{l_0}\texttt{)}^{l_1}$$

follows that also

$$(\hat{\mathsf{C}}, \hat{\mathsf{E}}, \hat{\rho}, \hat{\varphi}) \models_{bpe} t_0^{l_0} \tag{3.3}$$
$$\hat{\mathsf{C}}(l_0, bpe) \subseteq \hat{\rho}(x, bpe(x)) \tag{3.4}$$
$$\hat{\mathsf{C}}(l_0, bpe) \subseteq \hat{\mathsf{C}}(l_1, bpe) \tag{3.5}$$
$$\hat{\mathsf{E}}(l_0, bpe) \subseteq \hat{\mathsf{E}}(l_1, bpe) \tag{3.6}$$

holds by the clause [**setq**] of $\models$. Using the second and third assumption weakened is (3.5) and (3.6) to

$$\hat{\mathsf{C}}(l_0, bpe) \subseteq \hat{\mathsf{C}}(l_2, bpe) \tag{3.7}$$
$$\hat{\mathsf{E}}(l_0, bpe) \subseteq \hat{\mathsf{E}}(l_2, bpe). \tag{3.8}$$

The application of rule [**setq**] backwards on those two constraints, and (3.3) and (3.4) lets us deduce the proposition. All the other cases work accordingly as seen by inspecting the other clauses of $\models$. $\square$

One main insight towards the proof is, that the validity of the dynamic scope analysis of an expression carries over to its sub-expressions that are in the position of a possible hole of a evaluation context for the expression, and — even stronger — by the fact that such a sub-expression can be replaced in the expression by another valid one without violating its validity. With this result, the further proof of the correctness of a reduction step can concentrate on the possible redexes of all expressions; the following lemma then allows us to generalize the result to the big picture. This facility is known as *replacement lemma* in the realm of combinatory logic [HS86].

Lemma 4. *If* $(\hat{\mathsf{C}}, \hat{\mathsf{E}}, \hat{\rho}, \hat{\varphi}) \models_{bpe} \mathcal{E}[t_1^{l_1}]$ *where* $\mathcal{E}[t_1^{l_1}]$ *is carrying the label* l*, then there exists* bpe_0 *such that*

a) $(\hat{\mathsf{C}}, \hat{\mathsf{E}}, \hat{\rho}, \hat{\varphi}) \models_{bpe_0} t_1^{l_1}$ *holds,*

b) if also $(\hat{\mathsf{C}}, \hat{\mathsf{E}}, \hat{\rho}, \hat{\varphi}) \models_{bpe_0} t_2^{l_1}$ *holds then* $(\hat{\mathsf{C}}, \hat{\mathsf{E}}, \hat{\rho}, \hat{\varphi}) \models_{bpe} \mathcal{E}[t_2^{l_1}]$*.*

c) if also $\mathcal{E} \in \mathbf{VarContext}(x)$ *then* $bpe_0(x) = bpe(x)$*,*

d) if also $\mathcal{E} \in \mathbf{ExcContext}$ *then* $\hat{\mathsf{E}}(l_1, bpe_0) \subseteq \hat{\mathsf{E}}(l, me)$*, and*

Proof. The proof is done by structural induction over the evaluation context $\mathcal{E}$.
Case $\mathcal{E} \equiv [-]$. With bpe_0 chosen as

$$bpe_0 := bpe, \tag{3.9}$$

it immediately follows

a) by the assumption,

b) by the premise of d) since also (3.9) holds,

c) by (3.9), and also

d) by $l = l_1$ and (3.9).

Case $\mathcal{E}[t_1^{l_1}] \equiv \texttt{(if}\ \mathcal{E}'[t_1^{l_1}]\ e_3\ e_4\texttt{)}^l$ where $\mathcal{E}'[t_1^{l_1}]$ is carrying the label l_2. By the assumption one knows

$$(\hat{\mathsf{C}}, \hat{\mathsf{E}}, \hat{\rho}, \hat{\varphi}) \models_{bpe} \texttt{(if}\ \mathcal{E}'[t_1^{l_1}]\ e_3\ e_4\texttt{)}^l \tag{3.10}$$

and by using the clause [**if**] it follows

$$(\hat{\mathsf{C}}, \hat{\mathsf{E}}, \hat{\rho}, \hat{\varphi}) \models_{bpe} \mathcal{E}'[t_1^{l_1}] \ \wedge \tag{3.11}$$

$$\hat{\mathsf{E}}(l_2, bpe) \subseteq \hat{\mathsf{E}}(l, bpe) \ \wedge \tag{3.12}$$

$$\cdots . \tag{3.13}$$

From the application of the induction hypothesis on (3.11) we know that there exists a bpe_0 such that

$$(\hat{\mathsf{C}}, \hat{\mathsf{E}}, \hat{\rho}, \hat{\varphi}) \models_{bpe_0} t_1^{l_1}, \tag{3.14}$$

$$\text{if also } (\hat{\mathsf{C}}, \hat{\mathsf{E}}, \hat{\rho}, \hat{\varphi}) \models_{bpe_0} t_2^{l_1} \text{ holds then } (\hat{\mathsf{C}}, \hat{\mathsf{E}}, \hat{\rho}, \hat{\varphi}) \models_{bpe} \mathcal{E}'[t_2^{l_1}], \tag{3.15}$$

$$\text{if also } \mathcal{E}' \in \mathbf{VarContext}(x) \text{ then } bpe_0(x) = bpe(x), \text{ and} \tag{3.16}$$

$$\text{if also } \mathcal{E}' \in \mathbf{ExcContext} \text{ then } \hat{\mathsf{E}}(l_1, bpe_0) \subseteq \hat{\mathsf{E}}(l_2, me), \tag{3.17}$$

and finally that the propositions

a) since (3.14) holds,

b) by using the [**if**] clause backwards on (3.12), (3.13) and this time (3.15),

c) following from (3.16) and from the fact that if $\mathcal{E}' \in \mathbf{VarContext}(x)$ then also $\mathcal{E} \in \mathbf{VarContext}(x)$, and

d) following from (3.17), (3.12) and the transitivity of the subset relation, and from the fact that if $\mathcal{E}' \in \mathbf{ExcContext}$ then also $\mathcal{E} \in \mathbf{ExcContext}$

hold.

The proof of all the others cases is totally according. The propositions a) and b) always follow, since for all possible contexts of $\mathcal{E}$ there is always the validity needed for the sub-contexts also ensured by the acceptability relation $\models$. Part c) always holds, because $\models$ does not change the abstract binding in the memento environment for x in those contexts contained in $\mathbf{VarContext}(x)$. Accordingly $\models$ propagates the abstract exceptions further for the contexts in **ExcContext** enabling the proof for d). $\square$

All the needed prerequisites are available to proof the two main theorems. The first one will be the subject reduction for expressions under the reduction relation $\rightarrow$ saying that a valid dynamic scope analysis for an expression e and a correct approximation of the environments stay valid after one step with $\rightarrow$ for the resulting expression e' and the modified environments.

Theorem 1. *If* $(\hat{\mathsf{C}}, \hat{\mathsf{E}}, \hat{\rho}, \hat{\varphi}) \models_{bpe} \rho, \varphi, e$ *and* $\rho, \varphi, e \rightarrow \rho', \varphi', e'$ *then also* $(\hat{\mathsf{C}}, \hat{\mathsf{E}}, \hat{\rho}, \hat{\varphi}) \models_{bpe} \rho', \varphi', e'$.

Proof. Since the reduction relation $\rightarrow$ splits up in different reductions for several different reducible terms, we perform the proof by cases for those different reductions.

Case [**c**]: $\rho, \varphi, \mathcal{E}[c^l] \rightarrow \rho, \varphi, \mathcal{E}[\texttt{(const } c\texttt{)}^l]$. The assumption

$$(\hat{\mathsf{C}}, \hat{\mathsf{E}}, \hat{\rho}, \hat{\varphi}) \models_{bpe} \mathcal{E}[c^l]$$

leads to the existence of an bpe_0 such that

$$(\hat{\mathsf{C}}, \hat{\mathsf{E}}, \hat{\rho}, \hat{\varphi}) \models_{bpe_0} c^l$$

by applying Lemma 4. This is equivalent to

$$(\hat{\mathsf{C}}, \hat{\mathsf{E}}, \hat{\rho}, \hat{\varphi}) \models_{bpe_0} \texttt{(const } c\texttt{)}^l$$

which one gets by using rule [**c**] and [**const**] back and forth. Part b) of Lemma 4 shows us the proposition

$$(\hat{\mathsf{C}}, \hat{\mathsf{E}}, \hat{\rho}, \hat{\varphi}) \models_{bpe} \mathcal{E}[\texttt{(const } c^l\texttt{)}].$$

Case [**quote**]: $\rho, \varphi, \mathcal{E}[\texttt{(quote } s\texttt{)}^l] \rightarrow \rho, \varphi, \mathcal{E}[\texttt{(sym } s\texttt{)}^l]$. Analogous to case [**c**].

Case [**lambda**]: $\rho, \varphi, \mathcal{E}[\texttt{(lambda (}x\texttt{) } e\texttt{)}^l] \rightarrow \rho, \varphi, \mathcal{E}[\texttt{(proc (}x\texttt{) } e\texttt{)}^l]$. Analogous to case [**c**].

Case [**var**]: $\rho, \varphi, \mathcal{E}[\texttt{(bind } x\ v\ \mathcal{V}_x[x^l]\texttt{)}^{l_0}] \rightarrow \rho, \varphi, \mathcal{E}[\texttt{(bind } x\ v\ \mathcal{V}_x[v^l]\texttt{)}^{l_0}]$. By applying Lemma 4 a) to the assumption

$$(\hat{\mathsf{C}}, \hat{\mathsf{E}}, \hat{\rho}, \hat{\varphi}) \models_{bpe} \mathcal{E}[\texttt{(bind } x\ v\ \mathcal{V}_x[x^l]\texttt{)}^{l_0}] \tag{3.18}$$

we know that there exists an bpe_0 such that

$$(\hat{\mathsf{C}}, \hat{\mathsf{E}}, \hat{\rho}, \hat{\varphi}) \models_{bpe_0} \texttt{(bind } x\ v\ \mathcal{V}_x[x^l]\texttt{)}^{l_0}. \tag{3.19}$$

From the clause [**bind**] follows

$$(\hat{\mathsf{C}}, \hat{\mathsf{E}}, \hat{\rho}, \hat{\varphi}) \models_{bpe_1} \mathcal{V}_x[x^l] \tag{3.20}$$

$$\hat{\mathsf{C}}(l_2, bpe_1) \subseteq \hat{\mathsf{C}}(l_0, bpe_0) \tag{3.21}$$

$$\hat{\mathsf{E}}(l_2, bpe_1) \subseteq \hat{\mathsf{E}}(l_0, bpe_0) \tag{3.22}$$

$$v\, \mathcal{V}\, (\hat{\rho}(x, l_2), \hat{\mathsf{C}}) \tag{3.23}$$

where

$$l_2 \text{ is the label of } \mathcal{V}_x[x^l] \tag{3.24}$$

$$bpe_1 = bpe_0[x \mapsto l_2] \tag{3.25}$$

Using Lemma 4 a) and c) on (3.20) with the fact (3.25), we know that there exists an bpe_2 such that

$$(\hat{\mathsf{C}}, \hat{\mathsf{E}}, \hat{\rho}, \hat{\varphi}) \models_{bpe_2} x^l \tag{3.26}$$

$$bpe_2(x) = bpe_1(x) = l_2 \tag{3.27}$$

and further by using the clause [**var**] on (3.26) that

$$\hat{\rho}(x, l_2) \subseteq \hat{\mathsf{C}}(l, bpe_2). \tag{3.28}$$

With (3.23) and (3.26) it follows

$$v\,\mathcal{V}\,(\hat{\mathsf{C}}(l, bpe_2), \hat{\mathsf{C}}) \tag{3.29}$$

by Lemma 2 or — equivalent by Lemma 1 —

$$(\hat{\mathsf{C}}, \hat{\mathsf{E}}, \hat{\rho}, \hat{\varphi}) \models_{bpe_2} v^l. \tag{3.30}$$

With (3.20) and (3.30) Lemma 4 b) tells us

$$(\hat{\mathsf{C}}, \hat{\mathsf{E}}, \hat{\rho}, \hat{\varphi}) \models_{bpe_1} \mathcal{V}_x[v^l] \tag{3.31}$$

such that the conclusion is

$$(\hat{\mathsf{C}}, \hat{\mathsf{E}}, \hat{\rho}, \hat{\varphi}) \models_{bpe_0} \texttt{(bind}\ x\ v\ \mathcal{V}_x[v^l]\texttt{)}^{l_0} \tag{3.32}$$

by using the clause [**bind**] on (3.31) and (3.21) to (3.25). The result is

$$(\hat{\mathsf{C}}, \hat{\mathsf{E}}, \hat{\rho}, \hat{\varphi}) \models_{bpe} \mathcal{E}[\texttt{(bind}\ x\ v\ \mathcal{V}_x[v^l]\texttt{)}^{l_0}] \tag{3.33}$$

by Lemma 4 b) with (3.18) and (3.32).

Case [**setq**]: $\rho, \varphi, \mathcal{E}[\texttt{(bind}\ x\ v\ \mathcal{V}_x[\texttt{(setq}\ x\ v_0^{l_0}\texttt{)}^l]\texttt{)}^{l_1}] \to$
$\rho, \varphi, \mathcal{E}[\texttt{(bind}\ x\ v_0\ \mathcal{V}_x[v_0^l]\texttt{)}^{l_1}]$.

By applying Lemma 4 a) to the assumption

$$(\hat{\mathsf{C}}, \hat{\mathsf{E}}, \hat{\rho}, \hat{\varphi}) \models_{bpe} \mathcal{E}[\texttt{(bind}\ x\ v\ \mathcal{V}_x[\texttt{(setq}\ x\ v_0^{l_0}\texttt{)}^l]\texttt{)}^{l_1}] \tag{3.34}$$

There exists an me_0 such that

$$(\hat{\mathsf{C}}, \hat{\mathsf{E}}, \hat{\rho}, \hat{\varphi}) \models_{bpe_0} \texttt{(bind}\ x\ v\ \mathcal{V}_x[\texttt{(setq}\ x\ v_0^{l_0}\texttt{)}^l]\texttt{)}^{l_1}. \tag{3.35}$$

From the clause [**bind**] follows

$$(\hat{\mathsf{C}}, \hat{\mathsf{E}}, \hat{\rho}, \hat{\varphi}) \models_{bpe_1} \mathcal{V}_x[\texttt{(setq}\ x\ v_0^{l_0}\texttt{)}^l] \tag{3.36}$$

$$\hat{\mathsf{C}}(l_2, bpe_1) \subseteq \hat{\mathsf{C}}(l_1, bpe_0) \tag{3.37}$$

$$\hat{\mathsf{E}}(l_2, bpe_1) \subseteq \hat{\mathsf{E}}(l_1, bpe_0) \tag{3.38}$$

$$v\,\mathcal{V}\,(\hat{\rho}(x, l_2), \hat{\mathsf{C}}) \tag{3.39}$$

where

$$l_2 \text{ is the label of } \mathcal{V}_x[\texttt{(setq}\ x\ v_0^{l_0}\texttt{)}^l] \tag{3.40}$$

$$bpe_1 = bpe_0[x \mapsto l_2] \tag{3.41}$$

Using Lemma 4 a) and c) on (3.36) with the fact (3.41), there exists an bpe_2 such that

$$(\hat{\mathsf{C}}, \hat{\mathsf{E}}, \hat{\rho}, \hat{\varphi}) \models_{bpe_2} (\texttt{setq}\ x\ \ v_0^{l_0})^l \tag{3.42}$$

$$bpe_2(x) = bpe_1(x) = l_2 \tag{3.43}$$

and further by using the clause [**setq**] on (3.42) that

$$(\hat{\mathsf{C}}, \hat{\mathsf{E}}, \hat{\rho}, \hat{\varphi}) \models_{bpe_2} v_0^{l_0} \tag{3.44}$$

$$\hat{\mathsf{C}}(l_0, bpe_2) \subseteq \hat{\rho}(x, bpe_2(x)) \tag{3.45}$$

$$\hat{\mathsf{C}}(l_0, bpe_2) \subseteq \hat{\mathsf{C}}(l, bpe_2) \tag{3.46}$$

$$\hat{\mathsf{E}}(l_0, bpe_2) \subseteq \hat{\mathsf{E}}(l, bpe_2). \tag{3.47}$$

The validity of (3.44) is by Lemma 1 equivalent to

$$v_0\, \mathcal{V}\, (\hat{\mathsf{C}}(l_0, me_2), \hat{\mathsf{C}}) \tag{3.48}$$

which is weaker with (3.43), (3.45) and Lemma 2

$$v_0\, \mathcal{V}\, (\hat{\rho}(x, l_2), \hat{\mathsf{C}}). \tag{3.49}$$

By Lemma 3 with (3.46) and (3.47) the label of (3.44) can be changed to

$$(\hat{\mathsf{C}}, \hat{\mathsf{E}}, \hat{\rho}, \hat{\varphi}) \models_{bpe_2} v_0^{l} \tag{3.50}$$

With (3.36) and (3.50) Lemma 4 b) tells us

$$(\hat{\mathsf{C}}, \hat{\mathsf{E}}, \hat{\rho}, \hat{\varphi}) \models_{bpe_1} \mathcal{V}_x[v_0^l] \tag{3.51}$$

such that the conclusion is

$$(\hat{\mathsf{C}}, \hat{\mathsf{E}}, \hat{\rho}, \hat{\varphi}) \models_{bpe_0} (\texttt{bind}\ x\ \ v_0\ \ \mathcal{V}_x[v_0^l])^{l_1} \tag{3.52}$$

by using the clause [**bind**] on (3.51), (3.37), (3.38), and (3.49). The result is

$$(\hat{\mathsf{C}}, \hat{\mathsf{E}}, \hat{\rho}, \hat{\varphi}) \models_{bpe} \mathcal{E}[(\texttt{bind}\ x\ \ v_0\ \ \mathcal{V}_x[v_0^l])^{l_1}] \tag{3.53}$$

by Lemma 4 b) with (3.34) and (3.52).

Case [$\mathbf{var_{glob}}$]: $\rho, \varphi, \mathcal{V}_x[x^l] \to \rho, \varphi, \mathcal{V}_x[v^l]$
$x \in dom(\rho)$, $\rho(x) = v$

From the assumption

$$(\hat{\mathsf{C}}, \hat{\mathsf{E}}, \hat{\rho}, \hat{\varphi}) \models_{bpe} \mathcal{V}_x[x^l] \tag{3.54}$$

follows by Lemma 4 a] and c], that there exists a bpe_0, such that

$$(\hat{\mathsf{C}}, \hat{\mathsf{E}}, \hat{\rho}, \hat{\varphi}) \models_{bpe_0} x^l \tag{3.55}$$

$$bpe_0(x) = bpe(x) \tag{3.56}$$

is true. Using the [**var**]-rule on (3.55), it follows

$$\hat{\rho}(x, bpe_0(x)) \subseteq \hat{\mathsf{C}}(l, bpe_0). \tag{3.57}$$

On the other hand, applying the definition of $\models$ on the second assumption with the additional condition of this case

$$x \in dom(\rho),\ \rho(x) = v \tag{3.58}$$

gives us

$$v\, \mathcal{V}\, (\hat{\rho}(x, bpe(x)), \hat{\mathsf{C}}). \tag{3.59}$$

Because one knows (3.56), one can modify it by Lemma 2 together with (3.57) to

$$v\, \mathcal{V}\, (\hat{\mathsf{C}}(l, bpe_0), \hat{\mathsf{C}}) \tag{3.60}$$

which is in turn by Lemma 1 the same as

$$(\hat{\mathsf{C}}, \hat{\mathsf{E}}, \hat{\rho}, \hat{\varphi}) \models_{bpe_0} v^l. \tag{3.61}$$

The result finally follows from that, (3.54), and Lemma 4 c).
Case [$\mathbf{setq_{new}}$]: $\rho, \varphi, \mathcal{V}_x[\texttt{(setq}\ x\ v_0^{l_0}\texttt{)}^l] \to \rho[x \mapsto v_0], \varphi, \mathcal{V}_x[v_0^l]$. Having

$$(\hat{\mathsf{C}}, \hat{\mathsf{E}}, \hat{\rho}, \hat{\varphi}) \models_{bpe} \mathcal{V}_x[\texttt{(setq}\ x\ v_0^{l_0}\texttt{)}^l] \tag{3.62}$$
$$(\hat{\mathsf{C}}, \hat{\mathsf{E}}, \hat{\rho}, \hat{\varphi}) \models_{bpe} \rho, \varphi \tag{3.63}$$

concluding by Lemma 4 a) and c), that

$$(\hat{\mathsf{C}}, \hat{\mathsf{E}}, \hat{\rho}, \hat{\varphi}) \models_{bpe} \texttt{(setq}\ x\ v_0^{l_0}\texttt{)}^l \tag{3.64}$$
$$bpe_0(x) = bpe(x) \tag{3.65}$$

holds for some bpe_0. By the [**setq**]-rule this expands to

$$(\hat{\mathsf{C}}, \hat{\mathsf{E}}, \hat{\rho}, \hat{\varphi}) \models_{bpe_0} v_0^{l_0} \tag{3.66}$$
$$\hat{\mathsf{C}}(l_0, bpe_0) \subseteq \hat{\rho}(x, bpe_0(x)) \tag{3.67}$$
$$\hat{\mathsf{C}}(l_0, bpe_0) \subseteq \hat{\mathsf{C}}(l, bpe_0) \tag{3.68}$$
$$\hat{\mathsf{E}}(l_0, bpe_0) \subseteq \hat{\mathsf{E}}(l, bpe_0) \tag{3.69}$$

whereas the first proposition is the same as

$$v_0\, \mathcal{V}\, (\hat{\mathsf{C}}(l_0, bpe_0), \hat{\mathsf{C}}) \tag{3.70}$$

by Lemma 1, Using the fact (3.65), Lemma 2 tells us by the inclusion (3.67), that also

$$v_0\, \mathcal{V}\, (\hat{\rho}(x, bpe(x)), \hat{\mathsf{C}}) \tag{3.71}$$

holds. Combining this and the second assumption (3.63) using definition $\models$ gives us the first part of the result:

$$(\hat{\mathsf{C}}, \hat{\mathsf{E}}, \hat{\rho}, \hat{\varphi}) \models_{bpe} \rho[x \mapsto v_0], \varphi. \tag{3.72}$$

The second part is obtained by changing the label of (3.66) with Lemma 3, (3.68) and (3.68) to

$$(\hat{\mathsf{C}}, \hat{\mathsf{E}}, \hat{\rho}, \hat{\varphi}) \models_{bpe_0} v_0^l \tag{3.73}$$

and finally by applying Lemma 4 c) together with (3.62).

Case [$\mathbf{app_{lam}}$]: $\rho, \varphi, \mathcal{E}[\texttt{(}x_0^{l_0}\ \ e_1\texttt{)}^l] \rightarrow \rho, \varphi, \mathcal{E}[\texttt{(let}\ x_1\ \ e_1\ \ t_2^{l_2}\texttt{)}^l]$
$x_0 \in dom(\varphi)$, $\varphi(x_0) = \texttt{(proc (}x_1\texttt{)}\ e_2\texttt{)}$
By applying Lemma 4 a) on the the assumption

$$(\hat{\mathsf{C}}, \hat{\mathsf{E}}, \hat{\rho}, \hat{\varphi}) \models_{bpe} \mathcal{E}[\texttt{(}x_0^{l_0}\ \ e_1\texttt{)}^l] \tag{3.74}$$

there exists an bpe_0 such that

$$(\hat{\mathsf{C}}, \hat{\mathsf{E}}, \hat{\rho}, \hat{\varphi}) \models_{bpe_0} \texttt{(}x_0^{l_0}\ \ e_1\texttt{)}^l. \tag{3.75}$$

Using the definition of $\models$ on the two assumptions

$$(\hat{\mathsf{C}}, \hat{\mathsf{E}}, \hat{\rho}, \hat{\varphi}) \models_{bpe} \rho, \varphi \tag{3.76}$$

with the additional condition of this case

$$x_0 \in dom(\varphi),\ \varphi(x_0) = \texttt{(proc (}x_1\texttt{)}\ t_2^{l_2}\texttt{)} \tag{3.77}$$

yields

$$\texttt{(proc (}x_1\texttt{)}\ t_2^{l_2}\texttt{)}\ \mathcal{V}\,(\hat{\varphi}(x_0), \hat{\mathsf{C}}) \tag{3.78}$$

which is by the definition of $\mathcal{V}$ equivalent to

$$\texttt{(proc (}x_1\texttt{)}\ t_2^{l_2}\texttt{)} \in \hat{\varphi}(x_0). \tag{3.79}$$

The validity of (3.75) tells us by clause [**app**] that

$$\hat{\varphi}(x_0) \subseteq \hat{\mathsf{C}}(l_0, bpe_0) \tag{3.80}$$

and by the transitivity of (3.79) and that additionally

$$(\hat{\mathsf{C}}, \hat{\mathsf{E}}, \hat{\rho}, \hat{\varphi}) \models_{bpe_0} t_1^{l_1} \tag{3.81}$$
$$(\hat{\mathsf{C}}, \hat{\mathsf{E}}, \hat{\rho}, \hat{\varphi}) \models_{bpe_1} t_0^{l_b} \tag{3.82}$$
$$\hat{\mathsf{C}}(l_1, bpe_0) \subseteq \hat{\rho}(x_1, l_2) \tag{3.83}$$
$$\hat{\mathsf{E}}(l_1, bpe_0) \subseteq \hat{\mathsf{E}}(l, bpe_0) \tag{3.84}$$
$$\hat{\mathsf{C}}(l_2, bpe_1) \subseteq \hat{\mathsf{C}}(l, bpe_0) \tag{3.85}$$
$$\hat{\mathsf{E}}(l_2, bpe_1) \subseteq \hat{\mathsf{E}}(l, bpe_0) \tag{3.86}$$
where
$$bpe_1 = bpe_0[x_1 \mapsto l_2]). \tag{3.87}$$

But the propositions (3.81), (3.82), (3.83), (3.84), (3.85), (3.86) and (3.87) exactly the right hand side of the clause [**let**], so that it follows immediately

$$(\hat{\mathsf{C}}, \hat{\mathsf{E}}, \hat{\rho}, \hat{\varphi}) \models_{bpe_1} \texttt{(let } x_1 \;\; e_1 \;\; t_2^{l_2}\texttt{)}^l. \tag{3.88}$$

The result

$$(\hat{\mathsf{C}}, \hat{\mathsf{E}}, \hat{\rho}, \hat{\varphi}) \models_{bpe} \mathcal{E}[\texttt{(let } x_1 \;\; e_1 \;\; t_2^{l_2}\texttt{)}^l] \tag{3.89}$$

is just the conclusion of Lemma 4 b) on (3.74) and (3.88).

Case [**app$_{\mathbf{ind}}$**]: $\rho, \varphi, \mathcal{E}[\texttt{(}x_0^{l_0} \;\; e_1\texttt{)}^l] \to \rho, \varphi, \mathcal{E}[\texttt{(}s^{l_0} \;\; e_1\texttt{)}^l]$
$x_0 \in dom(\varphi)$, $\varphi(x_0) = \texttt{(sym } s\texttt{)}$

The assumption

$$(\hat{\mathsf{C}}, \hat{\mathsf{E}}, \hat{\rho}, \hat{\varphi}) \models_{bpe} \mathcal{E}[\texttt{(}x_0^{l_0} \;\; e_1\texttt{)}^l] \tag{3.90}$$

leads with Lemma 4 a) to

$$(\hat{\mathsf{C}}, \hat{\mathsf{E}}, \hat{\rho}, \hat{\varphi}) \models_{bpe} \texttt{(}x_0^{l_0} \;\; e_1\texttt{)}^l \tag{3.91}$$

for some *bpe*0. On the other hand, the condition of this case

$$x_0 \in dom(\varphi),\; \varphi(x_0) = \texttt{(sym } s\texttt{)} \tag{3.92}$$

together with the other assumption

$$(\hat{\mathsf{C}}, \hat{\mathsf{E}}, \hat{\rho}, \hat{\varphi}) \models_{bpe} \rho, \varphi \tag{3.93}$$

yields

$$\texttt{(sym } s\texttt{)} \; \mathcal{V} \; (\hat{\varphi}(x_0), \hat{\mathsf{C}}) \tag{3.94}$$

which is by the definition of $\mathcal{V}$ equivalent to

$$s \in \hat{\varphi}(x_0). \tag{3.95}$$

By applying rule [**app**] on (3.91), it follows

$$\hat{\varphi}(x_0) \subseteq \hat{\mathsf{C}}(l_0, bpe_0) \tag{3.96}$$

which in turn can be combined which the last but one proposition to

$$s \in \hat{\mathsf{C}}(l_0, bpe_0). \tag{3.97}$$

By that, the rule [**app**] shows the validity of more constraints, namely

$$(\hat{\mathsf{C}}, \hat{\mathsf{E}}, \hat{\rho}, \hat{\varphi}) \models_{bpe_0} \texttt{(}s^{l_0} \;\; t_1^{l_1}\texttt{)}^{l_s} \tag{3.98}$$

$$\hat{\mathsf{C}}(l_s, bpe_0) \subseteq \hat{\mathsf{C}}(l, bpe_0) \tag{3.99}$$

$$\hat{\mathsf{E}}(l_s, bpe_0) \subseteq \hat{\mathsf{E}}(l, bpe_0)) \tag{3.100}$$

which allows by Lemma 3 the change of the label to

$$(\hat{\mathsf{C}}, \hat{\mathsf{E}}, \hat{\rho}, \hat{\varphi}) \models_{bpe_0} (s^{l_0} \;\; t_1^{l_1})^l \tag{3.101}$$

from which we conclude the case by using Lemma 4 c) together with (3.90).

Case [**let**]: $\rho, \varphi, \mathcal{E}[(\texttt{let}\ x\ v_1^{l_1}\ \ e_2)^l] \to \rho, \varphi, \mathcal{E}[(\texttt{bind}\ x\ v_1\ \ e_2)^l]$. Lemma 4 a) used on the assumption

$$(\hat{\mathsf{C}}, \hat{\mathsf{E}}, \hat{\rho}, \hat{\varphi}) \models_{bpe} \mathcal{E}[(\texttt{let}\ x\ v_1^{l_1}\ \ e_2)^l] \tag{3.102}$$

tells us that there exists a bpe_0 such that

$$(\hat{\mathsf{C}}, \hat{\mathsf{E}}, \hat{\rho}, \hat{\varphi}) \models_{bpe_0} (\texttt{let}\ x\ v_1^{l_1}\ \ e_2)^l. \tag{3.103}$$

From the clause [**let**] it follows that

$$(\hat{\mathsf{C}}, \hat{\mathsf{E}}, \hat{\rho}, \hat{\varphi}) \models_{bpe_0} v_1^{l_1} \tag{3.104}$$
$$(\hat{\mathsf{C}}, \hat{\mathsf{E}}, \hat{\rho}, \hat{\varphi}) \models_{bpe_1} t_2^{l_2} \tag{3.105}$$
$$\hat{\mathsf{C}}(l_1, bpe_0) \subseteq \hat{\rho}(x, l_2) \tag{3.106}$$
$$\hat{\mathsf{E}}(l_1, bpe_0) \subseteq \hat{\mathsf{E}}(l, bpe_0) \tag{3.107}$$
$$\hat{\mathsf{C}}(l_2, bpe_1) \subseteq \hat{\mathsf{C}}(l, bpe_0) \tag{3.108}$$
$$\hat{\mathsf{E}}(l_2, bpe_1) \subseteq \hat{\mathsf{E}}(l, bpe_0) \tag{3.109}$$

where

$$bpe_1 = bpe_0[x \mapsto l_2], \tag{3.110}$$

hence it follows from (3.104) by using Lemma 1

$$v_1\, \mathcal{V}\, (\hat{\mathsf{C}}(l_1, bpe_0), \hat{\mathsf{C}}) \tag{3.111}$$

which leads to

$$v_1\, \mathcal{V}\, (\hat{\rho}(x, l_2), \hat{\mathsf{C}}) \tag{3.112}$$

by Lemma 2 and (3.106). From the [**bind**]-clause of the acceptability relation's definition with (3.105), (3.108), (3.109), (3.112), and (3.110), we get

$$(\hat{\mathsf{C}}, \hat{\mathsf{E}}, \hat{\rho}, \hat{\varphi}) \models_{bpe} (\texttt{bind}\ x\ \ v_1\ \ e_2)^l \tag{3.113}$$

which leads together with Lemma 4 b) and (3.102) to the result

$$(\hat{\mathsf{C}}, \hat{\mathsf{E}}, \hat{\rho}, \hat{\varphi}) \models_{bpe} \mathcal{E}[(\texttt{bind}\ x\ \ v_1\ \ e_2)^l] \tag{3.114}$$

Case [**bind**]: $\rho, \varphi, \mathcal{E}[(\texttt{bind}\ x\ \ v_0\ \ v_1^{l_1})^l] \to \rho, \varphi, \mathcal{E}[v_1^l]$. The conclusion of Lemma 4 a) on the assumption

$$(\hat{\mathsf{C}}, \hat{\mathsf{E}}, \hat{\rho}, \hat{\varphi}) \models_{bpe} \mathcal{E}[(\texttt{bind}\ x\ \ v_0\ \ v_1^{l_1})^l] \tag{3.115}$$

is, that there exists a bpe_0 such that

$$(\hat{\mathsf{C}}, \hat{\mathsf{E}}, \hat{\rho}, \hat{\varphi}) \models_{bpe_0} \texttt{(bind } x \; v_0 \; v_1^{l_1}\texttt{)}^l \tag{3.116}$$

holds. Using the clause [**bind**] on it, one knows, that

$$(\hat{\mathsf{C}}, \hat{\mathsf{E}}, \hat{\rho}, \hat{\varphi}) \models_{bpe_1} v_1^{l_1} \tag{3.117}$$
$$\hat{\mathsf{C}}(l_1, bpe_1) \subseteq \hat{\mathsf{C}}(l, bpe_0) \tag{3.118}$$
$$\hat{\mathsf{E}}(l_1, bpe_1) \subseteq \hat{\mathsf{E}}(l, bpe_0) \tag{3.119}$$
$$v\,\mathcal{V}\,(\hat{\rho}(x, l_1), \hat{\mathsf{C}}) \tag{3.120}$$
$$\textit{where}$$
$$bpe_1 = bpe_0[x \mapsto l_1] \tag{3.121}$$

holds. Lemma 1 on (3.117) gives us the different view

$$v\,\mathcal{V}\,(\hat{\mathsf{C}}(l_1, bpe_1), \hat{\mathsf{C}}) \tag{3.122}$$

on it. Lemma 2 and (3.118) weaken this to

$$v\,\mathcal{V}\,(\hat{\mathsf{C}}(l, bpe_0), \hat{\mathsf{C}}) \tag{3.123}$$

which is — again by Lemma 1 — the same as

$$(\hat{\mathsf{C}}, \hat{\mathsf{E}}, \hat{\rho}, \hat{\varphi}) \models_{bpe_0} v_1^l. \tag{3.124}$$

The conclusion

$$(\hat{\mathsf{C}}, \hat{\mathsf{E}}, \hat{\rho}, \hat{\varphi}) \models_{bpe} \mathcal{E}[v_1^l]. \tag{3.125}$$

is obtained by Lemma 4 b), (3.115) and (3.125).

Case [$\mathbf{if_1}$]: $\rho, \varphi, \mathcal{E}[\texttt{(if } v^{l_0} \; t_1^{l_1} \; e_2\texttt{)}^l] \to \rho, \varphi, \mathcal{E}[t_1^l]$, $v \neq \texttt{(sym nil)}$. From the assumption follows

$$(\hat{\mathsf{C}}, \hat{\mathsf{E}}, \hat{\rho}, \hat{\varphi}) \models_{bpe} \mathcal{E}[\texttt{(if } v^{l_0} \; t_1^{l_1} \; e_2\texttt{)}^l] \tag{3.126}$$

that there exists a bpe_0, such that

$$(\hat{\mathsf{C}}, \hat{\mathsf{E}}, \hat{\rho}, \hat{\varphi}) \models_{bpe_0} \texttt{(if } v^{l_0} \; t_1^{l_1} \; e_2\texttt{)}^l \tag{3.127}$$

holds, from which the [**if**]-clause tells us, that

$$(\hat{\mathsf{C}}, \hat{\mathsf{E}}, \hat{\rho}, \hat{\varphi}) \models_{bpe_0} v^{l_0} \tag{3.128}$$

is true. We further conclude by Lemma 1, the definition of $\mathcal{V}$, and the condition of this case

$$v \neq \texttt{(sym nil)} \tag{3.129}$$

that

$$\exists \hat{\mathsf{v}} \in \hat{\mathsf{C}}(l_0, bpe_0).\hat{\mathsf{v}} \neq \texttt{nil}. \tag{3.130}$$

holds. This enables to follow again from the [**if**]-clause the propositions

$$(\hat{\mathsf{C}}, \hat{\mathsf{E}}, \hat{\rho}, \hat{\varphi}) \models_{bpe} t_1^{l_1} \tag{3.131}$$
$$\hat{\mathsf{C}}(l_1, bpe_0) \subseteq \hat{\mathsf{C}}(l, bpe_0] \tag{3.132}$$
$$\hat{\mathsf{E}}(l_1, bpe_0) \subseteq \hat{\mathsf{E}}(l, bpe_0)) \tag{3.133}$$

which are exactly the preconditions for Lemma 3, such that

$$(\hat{\mathsf{C}}, \hat{\mathsf{E}}, \hat{\rho}, \hat{\varphi}) \models_{bpe} t_1^{l} \tag{3.134}$$

follows. The result is the consequence of Lemma 4 b) with (3.126) and (3.134).

Case [**if₂**]: $\rho, \varphi, \mathcal{E}[\texttt{(if } v^{l_0} \;\; e_2 \;\; t_2^{l_2}\texttt{)}^l] \to \rho, \varphi, \mathcal{E}[t_2^l]$, $v = \texttt{(sym nil)}$. The proof of this case works analogous to the previous one. Instead of the condition (3.129), we now know, that additionally

$$v = \texttt{(sym nil)} \tag{3.135}$$

is true. With this information available, this time the premise of the second implication of the [**if**]-clause becomes true, and it follows with the thus resulting proportions the result accordingly for t_2.

Case [**catch₁**]: $\rho, \varphi, \mathcal{E}[\texttt{(catch } v_0^{l_0} \;\; v_1^{l_1}\texttt{)}^l] \to \rho, \varphi, \mathcal{E}[v_1^l]$. The assumption

$$(\hat{\mathsf{C}}, \hat{\mathsf{E}}, \hat{\rho}, \hat{\varphi}) \models_{bpe} \mathcal{E}[\texttt{(catch } v_0^{l_0} \;\; v_1^{l_1}\texttt{)}^l] \tag{3.136}$$

leads to the existence of a bpe_0 with

$$(\hat{\mathsf{C}}, \hat{\mathsf{E}}, \hat{\rho}, \hat{\varphi}) \models_{bpe} \texttt{(catch } v_0^{l_0} \;\; v_1^{l_1}\texttt{)}^l \tag{3.137}$$

by Lemma 4 a). By the [**catch**] rule we also know, that

$$(\hat{\mathsf{C}}, \hat{\mathsf{E}}, \hat{\rho}, \hat{\varphi}) \models_{bpe_0} v_1^{l_1} \tag{3.138}$$
$$\hat{\mathsf{C}}(l_1, bpe_0) \subseteq \hat{\mathsf{C}}(l, bpe_0) \tag{3.139}$$
$$\hat{\mathsf{E}}(l_1, bpe_0) \subseteq \hat{\mathsf{E}}(l, bpe_0) \tag{3.140}$$

holds, hence by Lemma 3 also

$$(\hat{\mathsf{C}}, \hat{\mathsf{E}}, \hat{\rho}, \hat{\varphi}) \models_{bpe_0} v_1^{l} \tag{3.141}$$

is true. The case is proven by the following application of Lemma 4 b) together with (3.136).

Case [**catch₂**]: $\rho, \varphi, \mathcal{E}[\texttt{(catch } v_t^{l_t} \;\; \mathcal{X}[\texttt{(exc } v_0^{l_0} \;\; v_1^{l_1}\texttt{)}^{l_2}]\texttt{)}^l] \to \rho, \varphi, \mathcal{E}[v_1^l]$, $v_t = v_0$. The assumption

$$(\hat{\mathsf{C}}, \hat{\mathsf{E}}, \hat{\rho}, \hat{\varphi}) \models_{bpe} \mathcal{E}[\texttt{(catch } v_t^{l_t} \;\; \mathcal{X}[\texttt{(exc } v_0^{l_0} \;\; v_1^{l_1}\texttt{)}^{l_2}]\texttt{)}^l] \tag{3.142}$$

results with Lemma 4 a) to

$$(\hat{\mathsf{C}}, \hat{\mathsf{E}}, \hat{\rho}, \hat{\varphi}) \models_{bpe_0} \texttt{(catch } v_t^{l_t} \ \mathcal{X}[\texttt{(exc } v_0^{l_0} \ v_1^{l_1}\texttt{)}^{l_2}]\texttt{)}^l \tag{3.143}$$

for some bpe_0. For that, the [**catch**]-rule tells:

$$(\hat{\mathsf{C}}, \hat{\mathsf{E}}, \hat{\rho}, \hat{\varphi}) \models_{bpe_0} \mathcal{X}[\texttt{(exc } v_0^{l_0} \ v_1^{l_1}\texttt{)}^{l_2}] \tag{3.144}$$

$$\forall (l_4, bpe_4) \in \hat{\mathsf{E}}(l_3, bpe_0). \hat{\mathsf{C}}(l_4, bpe_4) \subseteq \hat{\mathsf{C}}(l, bpe_0) \tag{3.145}$$

where

$$l_3 \text{ is the label of } \mathcal{X}[\texttt{(exc } v_0^{l_0} \ v_1^{l_1}\texttt{)}^{l_2}] \tag{3.146}$$

on whose first proposition Lemma 2 a) and d) applies — because there is an exception context now — obtaining

$$(\hat{\mathsf{C}}, \hat{\mathsf{E}}, \hat{\rho}, \hat{\varphi}) \models_{bpe_1} \texttt{(exc } v_0^{l_0} \ v_1^{l_1}\texttt{)}^{l_2} \tag{3.147}$$

$$\hat{\mathsf{E}}(l_2, bpe_1) \subseteq \hat{\mathsf{E}}(l_3, bpe_0) \tag{3.148}$$

for some bpe_1, where the former conclusions is the same as

$$(\hat{\mathsf{C}}, \hat{\mathsf{E}}, \hat{\rho}, \hat{\varphi}) \models_{bpe_2} v_0^{l_0} \tag{3.149}$$

$$(\hat{\mathsf{C}}, \hat{\mathsf{E}}, \hat{\rho}, \hat{\varphi}) \models_{bpe_2} v_1^{l_1} \tag{3.150}$$

$$(l_1, bpe_2) \in \hat{\mathsf{E}}(l_2, bpe_1) \tag{3.151}$$

for some bpe_2 by the [**exc**]-rule, and the latter one (3.148) can be combined with (3.151) to

$$(l_1, bpe_2) \in \hat{\mathsf{E}}(l_3, bpe_0). \tag{3.152}$$

This enables (3.145) leading to

$$\hat{\mathsf{C}}(l_1, bpe_2) \subseteq \hat{\mathsf{C}}(l, bpe_0) \tag{3.153}$$

which allows as to write (3.150) differently by Lemma 1 as

$$v_1 \, \mathcal{V} \, (\hat{\mathsf{C}}(l_1, bpe_2), \hat{\mathsf{C}}) \tag{3.154}$$

and expanding it by Lemma 2 to

$$v_1 \, \mathcal{V} \, (\hat{\mathsf{C}}(l, bpe_0), \hat{\mathsf{C}}). \tag{3.155}$$

This is again by Lemma 1 equivalent to

$$(\hat{\mathsf{C}}, \hat{\mathsf{E}}, \hat{\rho}, \hat{\varphi}) \models_{bpe_0} v_1^l \tag{3.156}$$

which lets us end this case by finally using Lemma 4 b) together with (3.142).

Case [$\mathbf{catch_3}$]: $\rho, \varphi, \mathcal{E}[\texttt{(catch } v_t^{l_t} \ \mathcal{X}[\texttt{(exc } v_0^{l_0} \ v_1^{l_1}\texttt{)}^{l_2}]\texttt{)}^l] \rightarrow \rho, \varphi, \mathcal{E}[\texttt{(exc } v_0^{l_0} \ v_1^{l_1}\texttt{)}^l]$
$v_t \neq v_0$

Applying Lemma 4 a) on the assumption

$$(\hat{\mathsf{C}}, \hat{\mathsf{E}}, \hat{\rho}, \hat{\varphi}) \models_{bpe} \mathcal{E}[\texttt{(catch } v_t^{l_t} \ \mathcal{X}[\texttt{(exc } v_0^{l_0} \ v_1^{l_1}\texttt{)}^{l_2}]\texttt{)}^{l}] \tag{3.157}$$

yields, that there exists a bpe_0 such that

$$(\hat{\mathsf{C}}, \hat{\mathsf{E}}, \hat{\rho}, \hat{\varphi}) \models_{bpe_0} \texttt{(catch } v_t^{l_t} \ \mathcal{X}[\texttt{(exc } v_0^{l_0} \ v_1^{l_1}\texttt{)}^{l_2}]\texttt{)}^{l} \tag{3.158}$$

holds. From the [**catch**] clause, it follows that

$$(\hat{\mathsf{C}}, \hat{\mathsf{E}}, \hat{\rho}, \hat{\varphi}) \models_{bpe_0} \mathcal{X}[\texttt{(exc } v_0^{l_0} \ v_1^{l_1}\texttt{)}^{l_2}] \tag{3.159}$$

$$\hat{\mathsf{C}}(l_3, bpe_0) \subseteq \hat{\mathsf{C}}(l, bpe_0) \tag{3.160}$$

$$\hat{\mathsf{E}}(l_3, bpe) \subseteq \hat{\mathsf{E}}(l, bpe_0) \tag{3.161}$$

where

$$l_3 \text{ is the label of } \mathcal{X}[\texttt{(exc } v_0^{l_0} \ v_1^{l_1}\texttt{)}^{l_2}] \tag{3.162}$$

holds. Using again Lemma 4 a) and d) on (3.159) — part d) this time also applies since we are dealing now with an exception context —, we further know, that there also exists a bpe_1 such that

$$(\hat{\mathsf{C}}, \hat{\mathsf{E}}, \hat{\rho}, \hat{\varphi}) \models_{bpe_1} \texttt{(exc } v_0^{l_0} \ v_1^{l_1}\texttt{)}^{l_2} \tag{3.163}$$

$$\hat{\mathsf{E}}(l_2, bpe_1] \subseteq \hat{\mathsf{E}}(l_3, bpe_0) \tag{3.164}$$

is valid, and hence, by the [**exc**]-rule, that there also exists a bpe_2 with

$$(\hat{\mathsf{C}}, \hat{\mathsf{E}}, \hat{\rho}, \hat{\varphi}) \models_{bpe_2} v_0^{l_0} \tag{3.165}$$

$$(\hat{\mathsf{C}}, \hat{\mathsf{E}}, \hat{\rho}, \hat{\varphi}) \models_{bpe_2} v_1^{l_1} \tag{3.166}$$

$$(l_1, bpe_2) \in \hat{\mathsf{E}}(l_2, bpe_1). \tag{3.167}$$

From (3.164) and (3.167) follows

$$(l_1, bpe_2) \in \hat{\mathsf{E}}(l_3, bpe_0) \tag{3.168}$$

which results together with (3.165) and (3.166) and the [**exc**]-rule in

$$(\hat{\mathsf{C}}, \hat{\mathsf{E}}, \hat{\rho}, \hat{\varphi}) \models_{bpe_0} \texttt{(exc } v_0^{l_0} \ v_1^{l_1}\texttt{)}^{l_3}. \tag{3.169}$$

Lemma 3 together with (3.160) and (3.161) allows as to change its label to

$$(\hat{\mathsf{C}}, \hat{\mathsf{E}}, \hat{\rho}, \hat{\varphi}) \models_{bpe_0} \texttt{(exc } v_0^{l_0} \ v_1^{l_1}\texttt{)}^{l}, \tag{3.170}$$

hence, the result is by Lemma 4 b) and (3.157).

Case [**nocatch**]: $\rho, \varphi, \mathcal{X}[\texttt{(exc } v_0^{l_0} \ v_1^{l_1}\texttt{)}^{l_2}] \to \rho, \varphi, \texttt{(exc } v_0^{l_0} \ v_1^{l_1}\texttt{)}^{l_3}$
where l_3 is the label of $\mathcal{X}[\texttt{(exc } v_0^{l_0} \ v_1^{l_1}\texttt{)}^{l_2}]$

The assumption

$$(\hat{\mathsf{C}}, \hat{\mathsf{E}}, \hat{\rho}, \hat{\varphi}) \models_{bpe} \mathcal{X}[\texttt{(exc } v_0^{l_0} \ v_1^{l_1}\texttt{)}^{l_2}] \tag{3.171}$$

where

$$l_3 \text{ is the label of } \mathcal{X}[\texttt{(exc } v_0^{l_0} \ v_1^{l_1}\texttt{)}^{l_2}] \tag{3.172}$$

allows us to apply Lemma 4 a) and c), hence it exists a bpe_0, such that

$$(\hat{\mathsf{C}}, \hat{\mathsf{E}}, \hat{\rho}, \hat{\varphi}) \models_{bpe_0} \texttt{(exc } v_0^{l_0} \ v_1^{l_1}\texttt{)}^{l_2} \tag{3.173}$$

$$\hat{\mathsf{E}}(l_2, bpe_0) \subseteq \hat{\mathsf{E}}(l_3, bpe) \tag{3.174}$$

holds. Using the rule [**exc**] yields that there exists a bpe_1 such that

$$(\hat{\mathsf{C}}, \hat{\mathsf{E}}, \hat{\rho}, \hat{\varphi}) \models_{bpe_1} v_0^{l_0} \tag{3.175}$$

$$(\hat{\mathsf{C}}, \hat{\mathsf{E}}, \hat{\rho}, \hat{\varphi}) \models_{bpe_1} v_1^{l_1} \tag{3.176}$$

$$(l_1, bpe_1) \in \hat{\mathsf{E}}(l_2, bpe_0). \tag{3.177}$$

is true. It follows

$$(l_1, bpe_1) \in \hat{\mathsf{E}}(l_3, bpe). \tag{3.178}$$

from (3.174) and (3.177) which leads us to the claim

$$(\hat{\mathsf{C}}, \hat{\mathsf{E}}, \hat{\rho}, \hat{\varphi}) \models_{bpe} \texttt{(exc } v_0^{l_0} \ v_1^{l_1}\texttt{)}^{l_3} \tag{3.179}$$

by applying rule [**exc**] backwards together with (3.175) and (3.176).

Case [**cons**]: $\rho, \varphi, \mathcal{E}[\texttt{(cons } v_1^{l_1} \ v_2^{l_2}\texttt{)}^{l}] \to \rho, \varphi, \mathcal{E}[\texttt{(pair } v_1^{l_1} \ v_2^{l_2}\texttt{)}^{l}]$. With the assumption

$$(\hat{\mathsf{C}}, \hat{\mathsf{E}}, \hat{\rho}, \hat{\varphi}) \models_{bpe} \mathcal{E}[\texttt{(cons } v_1^{l_1} \ v_2^{l_2}\texttt{)}^{l}] \tag{3.180}$$

and Lemma 4 a), we conclude that there exists a bpe_0, such that

$$(\hat{\mathsf{C}}, \hat{\mathsf{E}}, \hat{\rho}, \hat{\varphi}) \models_{bpe_0} \texttt{(cons } v_1^{l_1} \ v_2^{l_2}\texttt{)}^{l} \tag{3.181}$$

holds. By the rule [**cons**] follow

$$(\hat{\mathsf{C}}, \hat{\mathsf{E}}, \hat{\rho}, \hat{\varphi}) \models_{bpe_0} v_1^{l_1} \tag{3.182}$$

$$(\hat{\mathsf{C}}, \hat{\mathsf{E}}, \hat{\rho}, \hat{\varphi}) \models_{bpe_0} v_2^{l_2} \tag{3.183}$$

$$(l_1, l_2, bpe_0) \in \hat{\mathsf{C}}(l, bpe_0), \tag{3.184}$$

hence by the rule [**pair**]

$$(\hat{\mathsf{C}}, \hat{\mathsf{E}}, \hat{\rho}, \hat{\varphi}) \models_{bpe_0} \texttt{(pair } v_1^{l_1} \ v_2^{l_2}\texttt{)}^{l}. \tag{3.185}$$

The case concludes by the following application of Lemma 4 b) with (3.180).

Case $[\mathbf{car_1}]$: $\rho, \varphi, \mathcal{E}[\texttt{(car (pair } v_1^{l_1} \; v_2^{l_2}\texttt{)}^{l_0}\texttt{)}^{l}] \to \rho, \varphi, \mathcal{E}[v_1^l]$. Using Lemma 4 a) on the assumption

$$(\hat{\mathsf{C}}, \hat{\mathsf{E}}, \hat{\rho}, \hat{\varphi}) \models_{bpe} \mathcal{E}[\texttt{(car (pair } v_1^{l_1} \; v_2^{l_2}\texttt{)}^{l_0}\texttt{)}^{l}] \tag{3.186}$$

yields, that there exists a bpe_0 such that

$$(\hat{\mathsf{C}}, \hat{\mathsf{E}}, \hat{\rho}, \hat{\varphi}) \models_{bpe_0} \texttt{(car (pair } v_1^{l_1} \; v_2^{l_2}\texttt{)}^{l_0}\texttt{)}^{l} \tag{3.187}$$

holds, and it follows from the $[\mathbf{car_1}]$-clause:

$$(\hat{\mathsf{C}}, \hat{\mathsf{E}}, \hat{\rho}, \hat{\varphi}) \models_{bpe_0} \texttt{(pair } v_1^{l_1} \; v_2^{l_2}\texttt{)}^{l_0} \tag{3.188}$$

$$\forall (l_1, l_2, bpe') \in \hat{\mathsf{C}}(l_0, bpe_0).\hat{\mathsf{C}}(l_1, bpe') \subseteq \hat{\mathsf{C}}(l, bpe_0) \tag{3.189}$$

$$\hat{\mathsf{E}}(l_0, bpe_0) \subseteq \hat{\mathsf{E}}(l, bpe_0). \tag{3.190}$$

By Lemma 1, (3.188) is the same as

$$\texttt{(pair } v_1^{l_1} \; v_2^{l_2}\texttt{)} \; \mathcal{V} \, (\hat{\mathsf{C}}(l_0, bpe_0), \hat{\mathsf{C}}) \tag{3.191}$$

which yields by definition of $\mathcal{V}$, that there exists a bpe_1 such that

$$(l_1, l_2, bpe_1) \in \hat{\mathsf{C}}(l_0, bpe_0) \tag{3.192}$$

$$v_1 \, \mathcal{V} \, (\hat{\mathsf{C}}(l_1, bpe_1), \hat{\mathsf{C}}) \tag{3.193}$$

$$v_2 \, \mathcal{V} \, (\hat{\mathsf{C}}(l_2, bpe_1), \hat{\mathsf{C}}) \tag{3.194}$$

holds. Combining (3.189) and (3.193) with Lemma 2 leads to

$$v_1 \, \mathcal{V} \, (\hat{\mathsf{C}}(l, bpe_0), \hat{\mathsf{C}}) \tag{3.195}$$

which is by Lemma 1 the same as

$$(\hat{\mathsf{C}}, \hat{\mathsf{E}}, \hat{\rho}, \hat{\varphi}) \models_{bpe_0} v_1^l. \tag{3.196}$$

The case concludes by now using Lemma 4 b) with (3.186).

Case $[\mathbf{car_2}]$: $\rho, \varphi, \mathcal{E}[\texttt{(car (sym nil)}^{l_0}\texttt{)}^{l}] \to \rho, \varphi, \mathcal{E}[\texttt{(sym nil)}^{l}]$. The assumption in this case

$$(\hat{\mathsf{C}}, \hat{\mathsf{E}}, \hat{\rho}, \hat{\varphi}) \models_{bpe} \mathcal{E}[\texttt{(car (sym nil)}^{l_0}\texttt{)}^{l}] \tag{3.197}$$

gives us by Lemma 4 a)

$$(\hat{\mathsf{C}}, \hat{\mathsf{E}}, \hat{\rho}, \hat{\varphi}) \models_{bpe_0} \texttt{(car (sym nil)}^{l_0}\texttt{)}^{l} \tag{3.198}$$

for some bpe_0, for which the clause $[\mathbf{car}]$ tells us

$$(\hat{\mathsf{C}}, \hat{\mathsf{E}}, \hat{\rho}, \hat{\varphi}) \models_{bpe_0} \texttt{(sym nil)}^{l_0} \tag{3.199}$$

$$\texttt{nil} \in \hat{\mathsf{C}}(l_0, bpe_0) \Rightarrow \texttt{nil} \in \hat{\mathsf{C}}(l, bpe_0). \tag{3.200}$$

Using the rule [**sym**] on (3.199) leads to

$$\texttt{nil} \in \hat{\mathsf{C}}(l_0, bpe_0) \tag{3.201}$$

which implies by (3.200)

$$\texttt{nil} \in \hat{\mathsf{C}}(l, bpe_0) \tag{3.202}$$

or equivalent again by the [**sym**]-rule

$$(\hat{\mathsf{C}}, \hat{\mathsf{E}}, \hat{\rho}, \hat{\varphi}) \models_{bpe_0} \texttt{(sym nil)}^l. \tag{3.203}$$

The result follows by Lemma 4 c) and (3.197).
Case [**funcall$_{\mathbf{lam}}$**]: $\rho, \varphi, \mathcal{E}[\texttt{(funcall (proc (}x_1\texttt{)}\ t_2^{l_2}\texttt{)}^{l_0}\ v_1^{l_1}\texttt{)}^l] \rightarrow$
$\rho, \varphi, \mathcal{E}[\texttt{(bind}\ x_1\ \ v_1\ \ e\texttt{)}^l]$
The assumption

$$(\hat{\mathsf{C}}, \hat{\mathsf{E}}, \hat{\rho}, \hat{\varphi}) \models_{bpe} \mathcal{E}[\texttt{(funcall (proc (}x_1\texttt{)}\ e\texttt{)}^{l_0}\ v_1^{l_1}\texttt{)}^l] \tag{3.204}$$

leads to

$$(\hat{\mathsf{C}}, \hat{\mathsf{E}}, \hat{\rho}, \hat{\varphi}) \models_{bpe_0} \texttt{(funcall (proc (}x_1\texttt{)}\ e\texttt{)}^{l_0}\ v_1^{l_1}\texttt{)}^l \tag{3.205}$$

for some bpe_0 by Lemma 4 a). Applying the clause [**funcall**] on it gives us

$$(\hat{\mathsf{C}}, \hat{\mathsf{E}}, \hat{\rho}, \hat{\varphi}) \models_{bpe_0} \texttt{(proc (}x_1\texttt{)}\ t_2^{l_2}\texttt{)}^{l_0} \tag{3.206}$$
$$(\hat{\mathsf{C}}, \hat{\mathsf{E}}, \hat{\rho}, \hat{\varphi}) \models_{bpe_0} v_1^{l_1}. \tag{3.207}$$

Since (3.206) is the same as

$$\texttt{(proc (}x_1\texttt{)}\ t_2^{l_2}\texttt{)}^{l_0} \in \hat{\mathsf{C}}(l_0, bpe_0] \tag{3.208}$$

by the [**proc**]-clause, the [**funcall**]-clause on (3.204) also results in

$$(\hat{\mathsf{C}}, \hat{\mathsf{E}}, \hat{\rho}, \hat{\varphi}) \models_{bpe_1} t_2^{l_2} \tag{3.209}$$
$$\hat{\mathsf{C}}(l_1, bpe_0) \subseteq \hat{\rho}(x_1, l_2) \tag{3.210}$$
$$\hat{\mathsf{C}}(l_2, bpe_1) \subseteq \hat{\mathsf{C}}(l, bpe_0) \tag{3.211}$$
$$\hat{\mathsf{E}}(l_2, bpe_1) \subseteq \hat{\mathsf{E}}(l, bpe_0) \tag{3.212}$$
where
$$bpe_1 = bpe[x_1 \mapsto l_2]. \tag{3.213}$$

If one applies on (3.207) Lemma 1, it is possible to use (3.210) with Lemma 2 to obtain

$$v_1\, \mathcal{V}\, (\hat{\rho}(x_1, l_2), \hat{\mathsf{C}}). \tag{3.214}$$

The propositions (3.209), (3.211), (3.212), (3.214), and (3.213) on the [**bind**]-clause result in

$$(\hat{\mathsf{C}}, \hat{\mathsf{E}}, \hat{\rho}, \hat{\varphi}) \models_{bpe_0} \texttt{(bind } x_1 \ v_1 \ t_2^{l_2}\texttt{)}^l \tag{3.215}$$

which gives us the conclusion by Lemma 4 c) and (3.204).

Case [**funcall$_{\mathbf{ind}}$**]: $\rho, \varphi, \mathcal{E}[\texttt{(funcall (sym } s_1\texttt{)}^{l_0} \ v_1^{l_1}\texttt{)}^l] \rightarrow$
$\rho, \varphi, \mathcal{E}[\texttt{(funcall } v_0^{l_0} \ v_1^{l_1}\texttt{)}^l]$
$s_1 \in dom(\varphi)$, $\varphi(s_1) = v_0$

The assumption

$$(\hat{\mathsf{C}}, \hat{\mathsf{E}}, \hat{\rho}, \hat{\varphi}) \models_{bpe} \mathcal{E}[\texttt{(funcall (sym } s_1\texttt{)}^{l_0} \ v_1^{l_1}\texttt{)}^l] \tag{3.216}$$

becomes

$$(\hat{\mathsf{C}}, \hat{\mathsf{E}}, \hat{\rho}, \hat{\varphi}) \models_{bpe_0} \texttt{(funcall (sym } s_1\texttt{)}^{l_0} \ v_1^{l_1}\texttt{)}^l \tag{3.217}$$

for some bpe_0 by Lemma 4 a). The clause [**funcall**] results in

$$(\hat{\mathsf{C}}, \hat{\mathsf{E}}, \hat{\rho}, \hat{\varphi}) \models_{bpe_0} \texttt{(sym } s_1\texttt{)}^{l_0} \tag{3.218}$$

$$(\hat{\mathsf{C}}, \hat{\mathsf{E}}, \hat{\rho}, \hat{\varphi}) \models_{bpe_0} v_1^{l_1} \tag{3.219}$$

for it, whereas the first proposition is equal to

$$s_1 \in \hat{\mathsf{C}}(l_0, bpe_0) \tag{3.220}$$

by the [**sym**]-clause. This enables to obtain

$$(\hat{\mathsf{C}}, \hat{\mathsf{E}}, \hat{\rho}, \hat{\varphi}) \models_{bpe_0} \texttt{(funcall } v^{t_0} \ v_1^{l_1}\texttt{)}^{l_s} \tag{3.221}$$

$$\hat{\mathsf{C}}(l_s, bpe_0) \subseteq \hat{\mathsf{C}}(l, bpe_0) \tag{3.222}$$

$$\hat{\mathsf{E}}(l_s, bpe_0) \subseteq \hat{\mathsf{E}}(l, bpe_0) \tag{3.223}$$

as further constraints by the [**funcall**]-rule on (3.216), since (3.220) and the additional condition of this case fulfill the second premise of the rule. Lemma 3 allows us to change the label of the form to

$$(\hat{\mathsf{C}}, \hat{\mathsf{E}}, \hat{\rho}, \hat{\varphi}) \models_{bpe_0} \texttt{(funcall } v^{t_0} \ v_1^{l_1}\texttt{)}^l, \tag{3.224}$$

hence it is possible to end the case by finally applying Lemma L:4 d] on that together with (3.216).

Case [**throw**]: $\rho, \varphi, \mathcal{E}[\texttt{(throw } v_1^{l_1} \ v_2^{l_2}\texttt{)}^l] \rightarrow \rho, \varphi, \mathcal{E}[\texttt{(exc } v_1^{l_1} \ v_2^{l_2}\texttt{)}^l]$. The assumption

$$(\hat{\mathsf{C}}, \hat{\mathsf{E}}, \hat{\rho}, \hat{\varphi}) \models_{bpe} \mathcal{E}[\texttt{(throw } v_1^{l_1} \ v_2^{l_2}\texttt{)}^l] \tag{3.225}$$

allows us to apply Lemma 4 a) such that it follows that there exists a bpe_0 with

$$(\hat{\mathsf{C}}, \hat{\mathsf{E}}, \hat{\rho}, \hat{\varphi}) \models_{bpe_0} \texttt{(throw } v_1^{l_1} \ v_2^{l_2}\texttt{)}^l \tag{3.226}$$

and we can conclude by the [**throw**]-rule, that

$$(\hat{\mathsf{C}}, \hat{\mathsf{E}}, \hat{\rho}, \hat{\varphi}) \models_{bpe_0} v_1^{l_1} \tag{3.227}$$

$$(\hat{\mathsf{C}}, \hat{\mathsf{E}}, \hat{\rho}, \hat{\varphi}) \models_{bpe_0} v_2^{l_2} \tag{3.228}$$

$$(l_2, bpe_0) \in \hat{\mathsf{E}}(l, bpe_0) \tag{3.229}$$

holds. Those conditions suffice the ones for [**exc**], therefore also

$$(\hat{\mathsf{C}}, \hat{\mathsf{E}}, \hat{\rho}, \hat{\varphi}) \models_{bpe_0} \texttt{(exc } v_1^{l_1}\ \ v_2^{l_2}\texttt{)}_0^l \tag{3.230}$$

and the result follows together with (3.225) by Lemma 4 b). □

Now having the subject reduction result for expression on hand, programs surely should also have this property. The second Theorem formulates this, adapting the former one to the reduction relation $\rightarrow_p$. It says again, that a dynamic scope analysis that is valid before a reduction step $\rightarrow_p$ stays valid after it.

Theorem 2. *If* $(\hat{\mathsf{C}}, \hat{\mathsf{E}}, \hat{\rho}, \hat{\varphi}) \models_{bpe} \rho, \varphi, p$ *and* $\rho, \varphi, p \rightarrow_d \rho', \varphi', p'$ *then also* $(\hat{\mathsf{C}}, \hat{\mathsf{E}}, \hat{\rho}, \hat{\varphi}) \models_{bpe} \rho', \varphi', p'$.

Proof. The different cases of $\rightarrow_d$ are considered separately.
Case [**defvar**]: $\rho, \varphi, \texttt{(defvar } x\ \ v_0^{l_0}\texttt{)}\ \ p \rightarrow_d \rho[x \mapsto v_0], \varphi, p$. The assumption

$$(\hat{\mathsf{C}}, \hat{\mathsf{E}}, \hat{\rho}, \hat{\varphi}) \models_{bpe} \texttt{(defvar } x\ \ v_0^{l_0}\texttt{)}\ \ p \tag{3.231}$$

lets us use the [**defvar**]-rule, hence it follows, that

$$(\hat{\mathsf{C}}, \hat{\mathsf{E}}, \hat{\rho}, \hat{\varphi}) \models_{bpe} v_0^{l_0} \tag{3.232}$$

$$\hat{\mathsf{C}}(l_0, bpe) \subseteq \hat{\rho}(x, bpe(x)) \tag{3.233}$$

$$(\hat{\mathsf{C}}, \hat{\mathsf{E}}, \hat{\rho}, \hat{\varphi}) \models_{bpe} p. \tag{3.234}$$

(3.234) is the first part of the result. Transforming (3.232) by Lemma 1 and combining it with (3.233) by Lemma 2 yields

$$v_0\, \mathcal{V}\, (\hat{\rho}(x, bpe(x)), \hat{\mathsf{C}}) \tag{3.235}$$

which results combined with the assumption

$$(\hat{\mathsf{C}}, \hat{\mathsf{E}}, \hat{\rho}, \hat{\varphi}) \models_{bpe} \rho, \varphi \tag{3.236}$$

in the second part of the result using the definition of $\models$.
Case [**defun**]: $\rho, \varphi, \texttt{(defun } x_0\ \ \texttt{(}x_1\texttt{)}\ \ e\texttt{)}\ \ p \rightarrow_d \rho, \varphi[x_0 \mapsto \texttt{(proc (}x_1\texttt{)}\ \ e\texttt{)}], p$. Having

$$(\hat{\mathsf{C}}, \hat{\mathsf{E}}, \hat{\rho}, \hat{\varphi}) \models_{bpe} \texttt{(defun } x_0\ \ \texttt{(}x_1\texttt{)}\ \ e\texttt{)}\ \ p \tag{3.237}$$

on hand leads to

$$\texttt{(proc (}x_1\texttt{)}\ e\texttt{)} \in \hat{\varphi}(x_0) \tag{3.238}$$

$$(\hat{\mathsf{C}}, \hat{\mathsf{E}}, \hat{\rho}, \hat{\varphi}) \models_{bpe} p \tag{3.239}$$

$$\tag{3.240}$$

by the [**defun**]-rule which on the one hand (3.237) easily combines with the first assumption and the definition of $\models$ to the first part of the result, and on the other hand (3.239) already is the second part of it.

Case [**exp**] a): $\rho, \varphi, e \rightarrow_d \rho', \varphi', e'$. Since the reduction $\rightarrow_d$ here is equal to a reduction step of $\rightarrow$, this case immediately follows by Theorem 1.

Case [**exp**] a): $\rho, \varphi, \texttt{(defvar}\ x\ e\texttt{)}\ a \rightarrow_d \rho', \varphi', \texttt{(defvar}\ x\ e'\texttt{)}\ a$. This follows easily by using the rule [**defvar**] from left to right, apply Theorem 1 on the subexpression e and the rule [**defvar**] backwards again, since the rule [**exp**] again reduces this reduction on a reduction step of $\rightarrow$. □

Chapter 4

Implementation

The previous chapter identified the valid dynamic scope analyses for a program, In other words, this is the mathematical version of exactly the information a compiler wants to use for the translation. A compiler usually does not work with paper and pencil. It needs an algorithmic representation of the process to determinate those information.

The general scheme to compute the information operates in two phases: the first phase generates a set of constraints for every program. Such a set of constraints expresses the relation between all the sets of abstract values that represent the different program points and variables. The second phase solves those constraints. Using a graph representation for problem of the the second phase is an effective way to tackle the problem.

The first section of this chapter explains how the constraint generation. The second section shows how to solve the constraints from the first phase left. The third section gives an overview of the implemented system.

4.1 Generating Constraints

Generated constraints should express the relations between the different components of the dynamic scope analysis. The solution of the constraints should correlate to the least dynamic scope analysis. In particular, it is important that there is only a finite number of constraints for every program. But what should those constraints look like? And what is the right way to generate the constraints for an arbitrary program?

Inspecting the acceptability relation for programs reveals the solution: there are only direct inclusion relations between abstract values and between abstract exceptions respectively. In addition, some of those relations hold only if a condition of some restricted form holds as well. *Constraints* of the form from Figure 4.1 are the only ones that are necessary to formulate such relations.

The variables $\hat{\mathsf{a}}$ are placeholders for sets of abstract objects — either

[**simple**]	$a \in \hat{\mathsf{a}}$
[**var**]	$\hat{\mathsf{a}}_0 \subseteq \hat{\mathsf{a}}_1$
[$\mathbf{cond_1}$]	$a \in \hat{\mathsf{a}} \implies \mathit{Constr}$
[$\mathbf{cond_2}$]	$\exists a_1 \in \hat{\mathsf{a}}.a_1 \neq a \implies \mathit{Constr}$
[$\mathbf{cond_3}$]	$\forall a_1 \in \hat{\mathsf{a}}.\mathit{Constr}(a_1)$

Figure 4.1: Constraints *Constr*.

abstract values $\hat{\mathsf{v}}$ or abstract exceptions $\hat{\mathsf{e}}$. Note that elements of the range of $\hat{\mathsf{C}}$, $\hat{\mathsf{E}}$, $\hat{\rho}$ and $\hat{\varphi}$ are from this type. Therefore, terms such as $\hat{\mathsf{C}}(l, bpe)$ show up in real constraints instead of $\hat{\mathsf{a}}$. Instead of a, single abstract values or exceptions occur in the generated constraints.

A [**simple**] constraint says that a certain abstract object a must be member of an abstract value or an abstract exception $\hat{\mathsf{a}}$. A constraint of the form [**var**] requires that the set of abstract objects $\hat{\mathsf{a}}_0$ is subset of the bigger set $\hat{\mathsf{a}}_1$. A constraint of the form [$\mathbf{cond_1}$] restricts some other constraints *Constr* upon the arrival of a condition: if the abstract object a is member of $\hat{\mathsf{a}}$. In conditional constraints of the form [$\mathbf{cond_2}$], the condition is slightly different: the condition will become true if there exists another abstract object a_1 in a set of abstract objects $\hat{\mathsf{a}}$ except the given one a. Conditions of the form [$\mathbf{cond_3}$] are more general. Every abstract objects a_1 that is member of $\hat{\mathsf{a}}$ enforces new constraints $\mathit{Constr}(a_1)$ depending on a_1.

The generation of such constraints is directly derived from the acceptability relation for programs $\models$ by inspecting the different clauses and transcribing the clauses to constraints. Figure 4.2 shows the Scheme code that generates constraints for `if` expressions derived from Figure 3.6.

The procedure `generate-constraints-cata-fun-if` is the procedure that is responsible for generating constraints for `if` expressions. This is not precisely true. The actual constraints depend on the different birthplace environments the handled `if` expression can show up under. When traversing a program to generate the constraints it is not known in advance with respect to which birthplace environments the constraint generation should happen. Generating constraints for *all* possible birthplace environments is a little bit tedious. This is the reason for not generating directly the constraints but for generating constraint-generating procedures that abstract over the yet unknown `bpe`s instead.

The different intermediate results in the `let` expression are completely according to the different sub-clauses of the [**if**] clause of Figure 3.6. The intermediate result `Test-constr` contains the constraints for the subexpres-

```
(define (generate-constraints-cata-fun-if node)
  (lambda (test-f e1-f elses-f)
    (let ((test (if-test node))
          (e1 (if-e1 node))
          (elses (if-elses node)))
      (lambda (exp)
        (lambda (bpe)
          (let ((test-constr (test-f bpe))
                (flowvar-test (make-flowvar/c test bpe))
                (conclusions-e1
                 (lambda ()
                   (let ((e1-constr (e1-f bpe)))
                     (set-of-constraints-union
                      (make-set-of-constraints
                       (make-constraint/var (make-flowvar/c e1 bpe)
                                            (make-flowvar/c exp bpe))
                       (make-constraint/var (make-flowvar/e e1 bpe)
                                            (make-flowvar/e exp bpe)))
                      e1-constr))))
                (conclusions-elses
                 (lambda ()
                   (let ((elses-constr (elses-f bpe)))
                     (set-of-constraints-union
                      (make-set-of-constraints
                       (make-constraint/var (make-flowvar/c elses bpe)
                                            (make-flowvar/c exp bpe))
                       (make-constraint/var (make-flowvar/e elses bpe)
                                            (make-flowvar/e exp bpe)))
                      elses-constr)))))
            (set-of-constraints-union
             (make-set-of-constraints
              (make-constraint/var (make-flowvar/e test bpe)
                                   (make-flowvar/e exp bpe))
              (make-constraint/cond-2 the-aval/sv-nil
                                      flowvar-test
                                      conclusions-e1)
              (make-constraint/cond-1 the-aval/sv-nil
                                      flowvar-test
                                      conclusions-elses))
             test-constr)))))))
```

Figure 4.2: Generating Constraints for `if` Expressions.

sion $t_0^{l_0}$ under the same birthplace environment as the instantiated `bpe`. `Conclusions-e1` contains the constraints for the premise of the conditional constraint that follow if an abstract value other than `nil` is in the resulting set for the conditional expression $t_0^{l_0}$. `Conclusion-elses` is a set of the analogous constraints for the condition constraint that emerge if the abstract symbol `nil` shows up for $t_0^{l_0}$. The result is the union of an [**var**] constraint that propagates exceptions, the two conditional constraints (a [$\mathbf{cond_2}$] constraint for the consequence and a [$\mathbf{cond_1}$] constraint for the alternative) and the constraints generated for the `test` expression.

When constructing the constraints for the premises of the conditional constraints, the resulting constraints of the conditional constraints are *not* immediately generated but their generation is delayed by putting it them a thunk. This is the general technique used to make the process of generating and solving constraints more "lazy". The generation of the premises of conditional constraints is always delayed by a thunk; the actual generation happens later on demand.

When processing a whole program, the first step is to generate the constraint generators for it. After this task is completed, the actual constraints are generated by instantiating the generated constraint generators with the toplevel birthplace environment. Additionally, one special precaution must take place because recursive calls can occur. To prevent an endless loop during the constraint generation the treatment of procedure calls is a variant of the practice described above: it uses memoization to remember the already processed procedure bodies. When a body is reentered, the generation of constraints stops.

4.2 Solving the Constraints

The generated constraints express the behavior of all valid dynamic scope analyses. To get the least dynamic scope analysis, the phase must compute the least solution of the generated constraints. A known way to do this is to consider the graph representation of this problem and to use a worklist algorithm on the graph to compute the least solution of the original constraints [AW93, JW95, WJ98].

The nodes of the graph correspond to the sets of abstract objects $\hat{\mathtt{a}}$ that already occurred (and will occur) while generating the constraints. A node has different components represented as a Scheme record. See Figure 4.3 for the actual Scheme code.

A node has data fields. They represent the actual set of abstract objects. For performance reasons, such a set is not kept in one field but is split up in two disjoint sets: the set associated with the field `new-data` represents those abstract objects that are new to this node with respect to the solving process, whereas the field `data` keeps those abstract objects that already

```
(define-record-type :node
  (make-node data new-data edges-new-data edges-all-data edges-rest)
  node?
  (data node-data set-node-data!)
  (new-data node-new-data set-node-new-data!)
  (edges-new-data node-edges-new-data set-node-edges-new-data!)
  (edges-all-data node-edges-all-data set-node-edges-all-data!)
  (edges-rest node-edges-rest set-node-edges-rest!))
```

Figure 4.3: Solving Constraints — Record `node`.

belong to the node.

The edges of a node represent constraints related to the node: those [**var**] or [**cond**] constraints which have the set of abstract objects $\hat{a}$ that the node is representing either on the left hand side (in case of a [**var**] constraint) or in the condition (in case of a [**cond**] constraint are possible members of the edges fields. Edges are either used to propagate data to other nodes (in case of [**var**] constraints) or to trigger the generation of new constraints — that is new nodes or new edges — (in case of [**cond**] constraints). Again, for performance reasons, the edges of a node are kept in different disjoint subsets: `Edges-new-data` are those edges which only new data effects, `edges-all-edges` are those which have to react on all data that belongs to the node, and `edges-rest` are edges that are not involved in the present state of the solving process.

The solving of the constraints roughly works as follows: an initial graph is constructed from the constraints obtained from the previous constraint generation phase. Data is propagated over the edges until no new data is moving over the graph and no new extensions of the graph are added (which can happen by triggering conditional constraints), Figure 4.4 shows a simplified version of the worklist algorithm that controls this process.

The global variable `*worklist*` always holds those nodes whose data still needs to be propagated over their edges.

The procedure `solve-constraints` is the main procedure: it initializes the worklist, builds up the initial graph using `expand-graph!` and starts the iteration process by calling `iterate-worklist`.

The procedure `expand-graph!` builds up the initial graph or expands an existing graph by applying new constraints. If for example a [**simple**] constraint is added, the procedure `add-abs-to-graph!` either adds an abstract object to the appropriate node or creates a new new, and additionally puts this node in the worklist, because new data arrived at the node. Similarly, a [**var**] constraint leads to a new edge.

The procedure `iterate-worklist` iterates over the worklist as long as there are still nodes in it. It removes a node from the worklist, first propa-

```
(define *worklist* #f)

(define (solve-constraints constraints)
  (init-worklist)
  (expand-graph! constraints)
  (iterate-worklist))

(define (init-worklist)
  (set! *worklist* (make-empty-worklist)))

(define (expand-graph! constraints)
  (set-of-constraints-for-each
   (lambda (constr)
     (cond
      ((constraint/simple? constr)
       (let ((abs (constraint/simple-abs constr))
             (flowvar (constraint/simple-flowvar constr)))
         (add-abs-to-graph! flowvar abs)))
      ((constraint/var? constr)
       (let ((flowvar0 (constraint/var-flowvar0 constr)))
         (add-edge-to-graph! flowvar0 constr)))
      ...))))

(define (iterate-worklist)
  (if (not (worklist-empty? *worklist*))
      (let ((node (worklist-remove! *worklist*)))
        (iterate-new-data node)
        (iterate-all-data node)
        (iterate-worklist))))

(define (iterate-new-data node)
  (let ((edges-new-data (node-edges-new-data node)))
    (if (not (set-of-constraints-empty? edges-new-data))
        (let ((constr (set-of-constraints-remove! edges-new-data))
              (new-data (node-new-data node))
              (edges-rest (node-edges-rest node)))
          (cond
           ((constraint/var? constr)
            (let ((flowvar1 (constraint/var-flowvar1 constr)))
              (add-new-data-to-graph! flowvar1 new-data))
            (set-of-constraints-adjoin-unsafe! constr edges-rest))
           ((constraint/cond-1? constr)
            (let ((abs (constraint/cond-1-abs constr))
                  (cos (constraint/cond-1-conclusions constr)))
              (if (set-of-abs-element? abs new-data)
                  (expand-graph! (cos))
                  (set-of-constraints-adjoin-unsafe! constr edges-rest))))
           ...
          (iterate-new-data node)))))

...
```

Figure 4.4: Solving Constraints — The Worklist Algorithm.

gates all of its new data — the data that recently arrived at the node and sits in the `new-data` file — over new data edges by calling `iterate-new-data` on that node, then propagates all of its data over the all data edges by calling `iterate-all-data` and starts over again.

The procedure `iterate-new-data` propagates all new data over the edges which are in the `edges-new-data` field of the actual node. The code sample shows an interesting case: if the actual node contains a [$\mathbf{cond_1}$] constraint and the conditional abstract object `abs` is member of `new-data`, the constraints of conclusions of the conditional constraint will be generated by calling the thunk which will add to graph.

On first sight, it does not seem clear that the worklist algorithm always terminates. But considering the fact that for every program there exists only a finite number of abstract values (cf. the definition of $\widehat{\mathbf{Val}}$) and that there is only a finite number of potential nodes (since there is only a finite number of program points, variables, and birthplace environments), an informal termination argument is possible: a finite number of data objects is propagated over a finite number of nodes. The process ends after a finite number of steps: at the latest at the point when every datum has arrived at every node.

The algorithm has an exponential worst case complexity with respect to the size of the analyzed program: the number of all possible birthplace environments is already exponential. However, for practical applications this is not a severe drawback. Wright and Jagannathan [WJ98] have done comparative measurements on a set of similar flow analyses. They came to the conclusion that an analysis with splitting — that is, a program point is considered several times — and exponential worst case complexity can outperform an analysis without splitting and polynomial worst case complexity. The reasonable argument is, that an analysis with splitting is more precise. The sets of abstract values are therefore smaller. Hence, a worklist algorithm has less data to propagate.

Demonstration of the Constraint Generation and Solving

This section presents a simple Emacs Lisp example and illustrates the constraint generation and solving. Figure 4.5 shows the example.

$$
\begin{array}{l}
\texttt{(defvar foo}\\
\quad\texttt{(lambda () x}^{1}\texttt{)}^{2}\texttt{)}\\
\\
\texttt{(let ((x 42}^{3}\texttt{))}\\
\quad\texttt{(funcall foo}^{4}\texttt{)}^{5}\texttt{)}^{6}
\end{array}
$$

Figure 4.5: Example.

`Foo` is a global variable bound to a procedure that returns the latest binding for the variable `x`. The second expression calls the procedure saved under the name `foo`, but prior to that binds the value `42` to the variable `x`. The result is `42`. This is obviously an example that only works with dynamic scoping. The numbers associated with the expressions are labels. They help to associate sets of abstract objects with expressions in this example.

The first phase, the generation of the constraints, leads to a set of constraints shown in Figure 4.6.

1. $\lambda_1 \in \hat{\mathsf{C}}(2, [])$
2. $\hat{\mathsf{C}}(2, []) \subseteq \hat{\rho}(\texttt{foo}, \diamond)$
3. $\omega \in \hat{\mathsf{C}}(3, [])$
4. $\hat{\mathsf{C}}(3, []) \subseteq \hat{\rho}(\texttt{x}, 6)$
5. $\hat{\rho}(\texttt{foo}, \diamond) \subseteq \hat{\mathsf{C}}(4, [\texttt{x} \rightarrow 6])$
6. $a \in \hat{\mathsf{C}}(4, [\texttt{x} \rightarrow 6]) \Longrightarrow \ldots$
7. $\hat{\mathsf{C}}(5, [\texttt{x} \rightarrow 6]) \subseteq \hat{\mathsf{C}}(6, [])$

Figure 4.6: Example — Generated Constraints.

Constraints 1 and 2 result from the definition of the variable `foo`: the first one states that the lambda-expression is a possible abstract value for the expression with label 2 under the top-level birthplace environment. The second constraint says, that everything for label two is also a possible abstract value for the variable `foo` introduced on top-level. Constraints 3 to 7 result from the `let` expression. Constraint 3 associates a constant to label 3. Constraint 4 says that all values for label 3 are possible values for the variable `x` introduced by the `let` expression with label 6. Constraint 5 connects the possible values for variable `foo` with the operator position of the procedure call at label 4. Constraint 6 is a conditional constraint of type [$\mathbf{cond_3}$]. Depending on what is member of the set for the operator position, different constraints will result. Constraint 7 links the body of the `let` expression labeled 5 with the whole `let` expression labeled with 6.

These constraints are the input of the following call to the procedure `solve-constraints` which uses the worklist algorithm to determine the solution. The first step what happens there is the building of the initial graph by `expand-graph!`. Figure 4.7 gives a rough idea who the resulting graph looks like.

A sets associated to a node shows all data that belongs to the node. To keep the picture of the graph simpler, the data is not split up in two sets as in the real implementation. Also, there is no distinction between different edges going out of an edge. An edge is drawn as an arrow ([**var**] constraint)

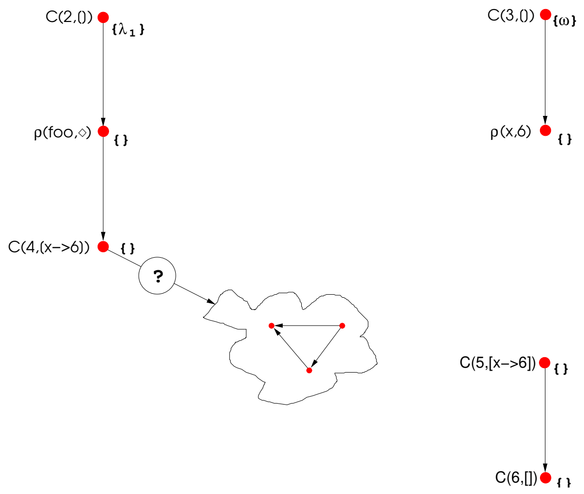

Figure 4.7: Example — Initial Graph.

or as a cloud ([**cond**] constraint) standing for the possible new parts of the graph that can be generated by triggering such an edge.

The `iterate-worklist` procedure continues by propagating the data over the graph. Supposing the algorithm first propagates the abstract value λ_1 representing the anonymous procedure over the graph, this abstract values first reaches the node for the variable `foo`, then the expression at the operator position labeled with 4. The test of the edges of this node triggers constraint 6 — the conditional constraint. As a consequence, new constraints are generated with the knowledge that procedure λ_1 is the new data at the node. Figure 4.8 shows the new constraints.

8. $\hat{\rho}(\mathtt{x}, 6) \subseteq \hat{\mathsf{C}}(1, [\mathtt{x} \to 6])$
9. $\hat{\mathsf{C}}(1, [\mathtt{x} \to 6]) \subseteq \hat{\mathsf{C}}(5, [\mathtt{x} \to 6])$

Figure 4.8: Example — New Constraints.

The new constraints originate from the body of the procedure. Constraint 8 connects the actual binding of the variable `x` from the `let` expression with label 6 with the body of procedure labeled with 1. Constraint 9 connects the results of the body with the result of the funcall of `foo`. Those new constraints are again inserted in the graph by the procedure `expand-graph!`. Figure 4.9 shows the resulting graph.

The worklist algorithm finally propagates the basic constant until it reaches the node representing set of abstract values for the final `let` ex-

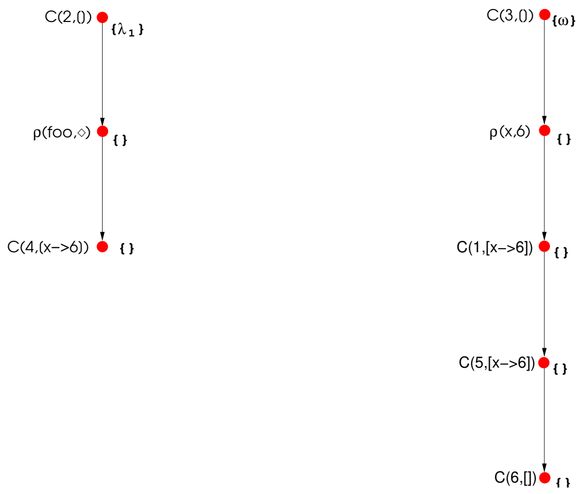

Figure 4.9: Example — Final Graph.

pression under the top level birthplace environment. It then terminates because no further new data has to run over the graph.

4.3 Overview of the System

The section gives a brief overview of the implemented system that emerged from the work of the thesis. The system analyzes (almost all) Emacs Lisp programs using Dynamic Scope Analysis and prints out annotated versions of the original programs to show the results. It is in about 5500 lines of Scheme code using the Scheme 48 system [KR95]. The implementation only uses Scheme features conforming to the latest Scheme report [KCR98]. The only external code is either library code that arose from the *Scheme Requests for Implementation* (SRFI) process[1] or taken from the first implementation of the project `el->scm` [BGHS00]. The system has the basic structure of a classical compiler with several passes. What follows are brief discussions how those passes are implemented.

Scanning and Parsing

The scanner and the parser stem from the original `el->scm` implementation. The scanner is an adaption of the Scheme 48 scanner. The parser is a hand-coded recursive descent parser. It produces a representation of the abstract syntax of the original program using Scheme records.

[1]The SRFIs are publicly accessible under `http://srfi.schemers.org`.

Generating Constraints

To generate the constraints using the methods of Section 4.1 the system traverses the abstract syntax obtained from the parser. The actual traversal is formulated as an instance of the catamorphism on the abstract syntax [BdM93].

Solving the Constraints

To solve the constraints, the system uses the worklist algorithm described in Section 4.2. A sets of abstract objects that occurs during the solving process is linked to the expression of the abstract syntax it belongs to. Doing this makes it easier for later passes to find the obtained information from the analysis associated with an expression..

Collecting the Information

The system also implements an emitter that determines the type of every variable binding — that is, whether a variable really needs to be dynamically or not. If it is only used in such a manner that the variable refers only to those birthplaces within lexical scope, a trivial translation to Scheme is possible. The *binding type rule* which decides to which type a variable reference belongs, is as follows:

$$\begin{gathered} x^{l_0} \text{ with static birthplace } l_1 \text{ is dynamic} \\ \textbf{iff} \\ \exists bpe.bpe(x) \neq l_1 \wedge \hat{\mathsf{C}}(l_0, bpe] \neq \emptyset. \end{gathered}$$

A variable is used dynamically if the abstract cache registers some abstract object for the variable under its label a *different* birthplace than the static one.

This pass is also implemented as an instance of the catamorphism over the abstract syntax. The system registers the determined information at the birthplaces of the variables.

Annotating the Variables

The information which the previous pass registers at the birthplaces has to be distributed to the various places where variables are referenced. Another instance of the catamorphism accomplishes this simple task.

Emitting the Annotated Code

A last instance of the catamorphism emits the program. The form of the output is regular Emacs Lisp code with annotations. Every variable with is used dynamically gets an annotation.

4.4 Sample Session

To demonstrate the system, this section shows a sample session analyzing a simple Emacs Lisp program. Consider the program shown in Figure 4.10 which serves as the sample input of the analysis.

```
(defun h3 (x)
  (let ((f (function (lambda() x))))
    (h4 f)))

(defun h4 (f)
  (let ((x 1))
    (funcall f)))

(h3 0)
```

Figure 4.10: Sample Session — Input.

The global procedure `h3` has one argument `x`. When applied, this procedure first creates a local procedure `f` that returns a variable `x` and then calls the procedure `h4` together with the new procedure `f`. The procedure `h4` wants a procedure `f` as argument. It first binds the variable `x` to the value 1 and then calls the passed procedure `f`. There is a starting call to `h3` with 0 as operand. Obviously, if an Emacs Lisp interpreter runs this program, the result is 1, because the passed procedure `f` returns the latest dynamically bound value of `x`, which is 1 by the `let` expression in `h4`.

Running the system on the program results in the output shown in Figure 4.11.

```
(defun h3 ((*unused* x))
  (let ((f (lambda ()
             (*dynamic* x))))
    (funcall 'h4 f)))

(defun h4 (f)
  (let (((*dynamic* x) 1))
    (funcall f)))

(funcall 'h3 0)
```

Figure 4.11: Sample Session — Output.

The system recognizes that the operand of the procedure `h3` is never used — hence its annotation `*unused*`. As already illustrated above, the function `f` of `h3` uses the variable `x` bound by the `let` expression in `h4` dynamically. The system identifies the dynamic usage and annotates `x` both in `h3` and `h4` with `*dynamic*`. An example for a variable that is statically used is `f` in procedure `h4`: the only point of reference to `f` is inside the the body of `h4`. The system does not annotate the variables that are used as with lexical scoping.

Chapter 5

Limitations, Future Work and Conclusion

5.1 Limitations

The implementation as accomplished for this thesis still has some limitations; not every Emacs Lisp program that works under GNU Emacs or XEmacs can be analyzed yet.

There are several constructs of the language that the parser does not understand and consequently the analysis does not handle. Similarly, there are features of Emacs Lisp that the analysis does not know about and therefore cannot properly analyze. The major shortcomings are as follows:

- The scanner, the parser and the representation of the abstract syntax for Emacs Lisp came directly from `el->scm`. The parser is a hand-coded recursive descent parser. It has no error recovery mechanism; In case of an malformed Emacs-Lisp program, it simply bails out.

- The parser understands only quoted symbols. All other quoted objects such as list or vector do not pass the parser.

- Emacs Lisp has a special form to define macros. Neither the parser nor the analysis identifies macros correctly.

- The analysis only handles the aggregate types pairs. Others, such as vectors, are not supported.

- Emacs Lisp manages symbols in special data structures. Most live in a global table the `obarray`. However, uninterned symbols also exist and behave differently. The analysis does not know how to handle uninterned symbols.

- Emacs Lisp also knows variables that are local to a text buffer of the editor. The dynamic scope analysis does not handle these variables specifically.

- The analysis does not separately trace effects, such as variable side effects or i/o effects. Because the evaluation order of arguments of a procedure call is fixed in Emacs Lisp and not fixed in Scheme, procedure calls cannot really be translated "idiomatically" from Emacs Lisp into Scheme.

5.2 Future Work

At the end of the migration from Emacs Lisp to Scheme, the end product should be a running Emacs with Scheme as extension language and with all packages working. To achieve the migration of the packages, the compiler must be completed. At least, the following tasks have to be tackled.

The static program analysis of this thesis must be made in to a full compiler. So far, as demonstrated in Chapter 4, the system is just a tool that does the analysis and outputs the annotated original program. An emitter which decides the types the types of bindings already exists as a prototype in the system. Code which recognizes name space conflicts and separates code and data is still missing. Specifically, in Emacs Lisp the symbol `nil` represents the empty list, the symbol itself and the boolean value false. Another challenge is the mapping of `nil` on the relevant Scheme value. The `el->scm` project already has a working emitter which uses Scheme macros to express the translated Emacs Lisp constructs. An overhaul of this emitter is required.

The analysis knows only the most essential Emacs Lisp primitives. However, the typical XEmacs distribution contains over 1500 primitives coded in C. As a consequence, the analysis cannot accurately examine most of the built-in functions. Nevertheless, in the end it must deal with all of them and know their abstract behavior. They could be recoded in Emacs Lisp (if at all possible), or they could be incorporated in the analysis, directly or indirectly, by a description language.

It will not be possible to translate all the existing packages with their over 500,000 lines of Emacs Lisp code at once. Rather, it must be possible to analyze and translate packages separately with still handling dependencies between the packages. Cormac Flanagan and Matthias Felleisen showed how to develop such a componential flow analysis [FF97].

5.3 Conclusion

With the help of static program analysis techniques source-to-source translations of high-level programming languages are possible. Moreover, they even enable idiomatic translation — a translation that conserves the appearance and the programmer's intention of the original program — as this thesis demonstrates with the application of Dynamic Scope Analysis on a compiler for Emacs Lisp.

Dynamic Scope Analysis is a new constraint-based flow analysis developed that gains those information from Emacs Lisp programs needed for an idiomatic translation into higher-order programming languages with lexical scoping such as Scheme. It provides, among other things, information about the nature of the program's variable bindings to allow the compiler to eliminate unnecessary instances of dynamic binding in the target language. Key idea is to analyze program points differently for different variable bindings that occur at each program point.

Dynamic Scope Analysis has a formal specification and a correctness proof. This provides the programmer of an analysis with the right way to design its implementation. The derivation of constraints from the formal specification is easy. The programmer has the guaranty that those constraints represent a correct solution.

A necessary prerequisite for the formal correctness proof was a the formulation of a syntax and semantics for Emacs Lisp. The semantics presented here is the first formalization of the semantics of Emacs Lisp. Stating the semantics as a reduction semantics with evaluation contexts was a big win. Evaluation contexts are a perfect way to describe dynamic scoping.

Bibliography

[AW93] Alexander Aiken and Edward Wimmers. Type inclusion constraints and type inference. In *Proceedings of the FPCA 1993*, pages 31–41, 1993.

[Bar84] Henk P. Barendregt. *The Lambda Calculus: Its Syntax and Semantics, Revised edition*. North Holland, 1984.

[BdM93] Richard S. Bird and Oege de Moor. Solving optimisation problems with catamorphisms. In R. S. Bird, C. C. Morgan, and J. C. P. Woodcock, editors, *Proceedings 2nd Int. Conf. on Maths. of Program Construction, MPC'92, Oxford, UK, 29 June – 3 July 1992*, volume 669 of *Lecture Notes in Computer Science*, pages 45–66. Springer-Verlag, Berlin, 1993.

[BGHS00] Peter Biber, Martin Gasbichler, Johannes Hirche, and Michael Sperber. `el->scm`. unpublished manuscript, Wilhelm-Schickard-Institut, Tübingen, February 2000.

[Bib00] Peter Biber. Eine kurz drauflosgehackte Flussanalyse für Elisp. unpublished manuscript, Wilhelm-Schickard-Institut, Tübingen, February 2000.

[Bir93] Matthew Birkholz. Emacs Lisp in Edwin Scheme. Artificial Intelligence Lab Report TR-1451, MIT, September 1993.

[Bot00] Per Bothner. JEmacs: The Java/Scheme-based Emacs. In *Proceedings of the USENIX Annual 2000 Technical Conference — Freenix Refereed Track*, San Diego, CA, USA, June 2000.

[CC77] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction of approximation of fixed points. In *Proceedings of the 4th ACM Symposium on Principles of Programming Languages, Los Angeles*, pages 238–252, New York, NY, 1977. ACM.

[CF58] Haskell B. Curry and Robert Feys. *Combinatory Logic*, volume I. North-Holland, Amsterdam, 1958.

[EHM+99] Peter Harry Eidorff, Fritz Henglein, Christian Mossin, Henning Niss, Morten Heine Sørensen, and Mads Tofte. AnnoDomini: from type theory to Year 2000 conversion tool. In ACM, editor, *POPL '99. Proceedings of the 26th ACM SIGPLAN-SIGACT on Principles of programming languages, January 20–22, 1999, San Antonio, TX*, ACM SIGPLAN Notices, pages 1–14, New York, NY, USA, 1999. ACM Press.

[Fel91] Matthias Felleisen. On the expressive power of programming languages. In *Science of Computer Programming*, volume 17, pages 35–75, 1991. Preliminary version in: *Proc. European Symposium on Programming*, Lecture Notes in Computer Science, 432. Springer-Verlag (1990), 134–151.

[FF86] Matthias Felleisen and Daniel P. Friedman. Control operators, the SECD-machine, and the λ-calculus. In Martin Wirsing, editor, *Formal Description of Programming Concepts III*, pages 193–217. North-Holland, 1986.

[FF97] Cormac Flanagan and Matthias Felleisen. Componential set-based analysis. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI-97)*, volume 32, 5 of *ACM SIGPLAN Notices*, pages 235–248, New York, June 15–18 1997. ACM Press.

[Gos81] James Gosling. UNIX EMACS. Report, Massachusetts Institute of Technology, Artificial Intelligence Laboratory, Cambridge, MA, USA, December 1981.

[Gre80] Bernard S. Greenberg. Prose and CONS — Multics Emacs: A commercial text-processing system in LISP. In *Conference Record of the 1980 LISP Conference, Stanford University*, New York, NY 10036, USA, 1980. ACM Press.

[GS90] Carl A. Gunter and Dana S. Scott. Semantic domains. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B: Formal Models and Semantics, chapter 8, pages 633–674. North-Holland, New York, NY, 1990.

[GUI00] Guile Emacs. `http://gemacs.sourceforge.net/`, 2000. December 5.

[Hei94] Nevin Heintze. Set-based analysis of ML programs. In *ACM Conference on Lisp and Functional Programming*, pages 306–317, 1994.

[HS86] Roger Hindley and Jonathan Seldin. *Introduction or Combinators and Lambda Calculus*, volume 1 of *London Mathematical Society Student Texts*. Cambridge University Press, 1986.

[HS97] Robert Harper and Chris Stone. An interpretation of Standard ML in type theory. Technical Report CMU–CS–97–147, Pittsburgh, PA, June 1997. (Also published as Fox Memorandum CMU–CS–FOX–97–01.).

[JW95] Suresh Jagannathan and Stephen Weeks. A Unified Treatment of Flow Analysis in Higher-Order Languages. In *POPL*, 1995.

[Kah87] Gilles Kahn. Natural semantics. In *Proceedings of the Symposium on Theoretical Aspects of Computer Science*, pages 22–39, 1987.

[KCR98] Richard Kelsey, William Clinger, and Jonathan Rees. Revised^5 report on the algorithmic language Scheme. 11(1):7–105, 1998. Also appears in ACM SIGPLAN Notices 33(9), September 1998. Available electronically as `http://www.neci.nj.nec.com/homepages/kelsey/r5rs.ps.gz`.

[KR95] Richard A. Kelsey and Jonathan A. Rees. A tractable Scheme implementation. 7(4):315–335, 1995.

[LHJ95] Sheng Liang, Paul Hudak, and Mark Jones. Monad transformers and modular interpreters. In *Conference Record of POPL '94: 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Francisco, California*, pages 333–343. ACM, January 1995.

[LLStG85] Bil Lewis, Dan LaLiberte, Richard Stallman, and the GNU Manual Group. Gnu emacs lisp reference manual. `http://www.gnu.org/manual/elisp-manual-20-2.5/elisp.html`, 1785.

[McC81] John McCarthy. History of lisp. In Richard Wexelblat, editor, *History of Programming Languages*. Academic Press, 1981.

[Mos90] Peter D. Mosses. Denotational semantics. In J. van Leewen, editor, *Handbook of Theoretical Computer Science*, volume B: Formal Models and Semantics, chapter 11, pages 577–631. The MIT Press, New York, NY, 1990.

[Mos99] Peter D. Mosses. Foundations of modular sos (extended abstract). In Miroslaw Kutylowski, Leszek Pacholski, and Tomasz Wierzbicki, editors, *Mathematical Foundations of Computer*

Science 1999, 24th International Symposium, MFCS'99, volume 1672 of *Lecture Notes in Computer Science*, pages 70–80. Springer, 1999.

[MP88] John C. Mitchell and Gordon D. Plotkin. Abstact types have existantial type. In *ACM Transcations on Programmin Languages and Systems*, volume 10, pages 470–502, July 1988.

[MT91] Robin Milner and Mats Tofte. Co-induction in relational semantics. *Theoretical Computer Science*, 87:209–220, 1991.

[Myc81] Alan Mycroft. *Abstract Interpretation and Optimising Transformations for Applicative Programs.* PhD thesis, University of Edinburgh, United Kingdom, 1981.

[NN97] Flemming Nielson and Hanne Riis Nielson. Infinitary control flow analysis: a collecting semantics for closure analysis. In *Proc. POPL'97*, pages 332–345. ACM Press, 1997.

[NNH99] Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. *Principles of Program Analysis.* Springer-Verlag Berlin, 1999.

[Pit00] Andrew M. Pitts. Operational semantics and program equivalence, 2000.

[Plo81] Gordon D. Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN-19, Computer Science Department, Aarhus University, Aarhus, Denmark, September 1981.

[Que96] Christian Queinnec. *Lisp in Small Pieces.* Cambridge University Press, 1996.

[RFT99] G. Ramalingam, John Field, and Frank Tip. Aggregate structure identification and its application to program analysis. In ACM, editor, *POPL '99. Proceedings of the 26th ACM SIGPLAN-SIGACT on Principles of programming languages, January 20–22, 1999, San Antonio, TX*, ACM SIGPLAN Notices, pages 119–132, New York, NY, USA, 1999. ACM Press.

[SG93] Guy L. Steele Jr. and Richard P. Gabriel. The evolution of lisp. In *Second ACM SIGPLAN History of Programming Languages Conference*, April 93.

[Shi88] Olin Shivers. Control flow analysis in scheme. *ACM SIGPLAN Notices*, 23(7):164–174, July 1988.

[Shi91] Olin Shivers. *Control-Flow Analysis of Higher-Order Languages.* PhD thesis, Carnegie-Mellon University, May 1991.

[Sta79] Richard M. Stallman. EMACS: The extensible, customizable, self-documenting display editor. Technical Report AIM-519A, Massachusetts Institute of Technology, June 1979.

[WC98] Mitchell Wand and William D. Clinger. Set constraints for destructive array update optimization. In *Proceedings of ICCL'98*, Chicago, Illinois, May 1998.

[WF91] Andrew K. Wright and Matthias Felleisen. A syntactic approach to type soundness. Technical Report 91-160, Rice University, April 1991. Final version in *Information and Computation* **115** (1), 1994, 38–94.

[WF94] Andrew K. Wright and Matthias Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, 1994. Preliminary version in Rice TR 91-160.

[Win99] Ben Wing. XEmacs Lisp Reference Manual. `ftp://ftp.xemacs.org/pub/xemacs/docs/a4/lispref-a4.pdf.gz`, May 1999. Version 3.4.

[WJ98] Andrew K. Wright and Suresh Jagannathan. Polymorphic splitting: an effective polyvariant flow analysis. *ACM Transactions on Programming Languages and Systems*, 20(1):166–207, January 1998.

[WS99] Mitchell Wand and Igor Siveroni. Constraint systems for useless variable elimination. In *Proceedings of 26th ACM Symposium on Priciples of Programming Languages*, Houston, TX, 1999.

Index