Down with Emacs Lisp: Dynamic Scope Analysis

Matthias Neubauer
Institut für Informatik
Universität Freiburg
neubauer@informatik.uni-freiburg.de

Michael Sperber
Wilhelm-Schickard-Institut für Informatik
Universität Tübingen
sperber@informatik.uni-tuebingen.de

ABSTRACT

It is possible to translate code written in Emacs Lisp or another Lisp dialect which uses dynamic scoping to a more modern programming language with lexical scoping while largely preserving structure and readability of the code. The biggest obstacle to such an idiomatic translation from Emacs Lisp is the translation of dynamic binding into suitable instances of lexical binding: Many binding constructs in real programs in fact exhibit identical behavior under both dynamic and lexical binding. An idiomatic translation needs to detect as many of these binding constructs as possible and convert them into lexical binding constructs in the target language to achieve readability and efficiency of the target code.

The basic prerequisite for such an idiomatic translation is thus a *dynamic scope analysis* which associates variable occurrences with binding constructs. We present such an analysis. It is an application of the Nielson/Nielson framework for flow analysis to a semantics for dynamic binding akin to Moreau's. Its implementation handles a substantial portion of Emacs Lisp, has been applied to realistic Emacs Lisp code, and is highly accurate and reasonably efficient in practice.

1. MIGRATING EMACS LISP

Emacs Lisp [16, 29] is a popular programming language for a considerable number of desktop applications which run within the Emacs editor or one of its variants. The actively maintained code base measures at around 1,000,000 loc[1]. As the Emacs Lisp code base is growing, the language is showing its age: It lacks important concepts from modern functional programming practice as well as provisions for large-scale modularity. Its implementations are slow compared to mainstream implementations of other Lisp dialects. Moreover, the development of both Emacs dialects places comparatively little focus on significant improvements of the Emacs Lisp interpreter.

On the other hand, recent years have seen the advent of a large number of *extension language* implementations of full programming languages suitable for the inclusion in application software. Specifically, several current Scheme implementations are technologically much better suited as an extension language for Emacs than Emacs Lisp itself. In fact, the official long-range plan for GNU Emacs is to replace the Emacs Lisp substrate with Guile, also a Scheme implementation [28]. The work presented here is part of a different, independent effort to do the same for XEmacs, a variant of GNU Emacs which also uses Emacs Lisp as its extension language.

Replacing such a central part of an application like XEmacs presents difficult pragmatic problems: It is not feasible to reimplement the entire Emacs Lisp code base by hand. Thus, a successful migration requires at least the following ingredients:

- Emacs Lisp code must continue to run unchanged for a transitory period.
- An automatic tool translates Emacs Lisp code into the language of the new substrate, and it must produce maintainable code.

Whereas the first of these ingredients is not particularly hard to implement (either by keeping the old Emacs Lisp implementation around or by re-implementing an Emacs Lisp engine in the new substrate), the second is more difficult. Even though a direct one-to-one translation of Emacs Lisp into a modern latently-typed functional language is straightforward by using dynamic assignment or dynamic-environment passing to implement dynamic scoping, it does not result in maintainable output code: Users of modern functional languages use dynamic binding only in very limited contexts such as exception handling or parameterization. As it turns out, the situation is not much different for Emacs Lisp users: For many `let`s and other binding constructs in real Emacs Lisp code, dynamic scope and lexical scope are *identical*! Consequently, a good "idiomatic" translation of Emacs Lisp into, say, Scheme, should convert these binding constructs into the corresponding lexical binding constructs of the target substrate.

The only problem is to *recognize* these binding constructs, or rather, distinguish those where the programmer "meant" dynamic scope from those where she "meant" lexical scope. Since with dynamic scope, bindings travel through the program execution much as values do, this requires a proper

[1] The XEmacs package collection which includes many popular add-ons and applications currently contains more than 700,000 loc.

ICFP'01, September 3-5, 2001, Florence, Italy.

```
(let* ((filename (expand-file-name filename))
       (file (file-name-nondirectory filename))
       (dir  (file-name-directory    filename))
       (comp (file-name-all-completions file dir))
       newest)
  (while comp
    (setq file (concat dir (car comp))
          comp (cdr comp))
    (if (and (backup-file-name-p file)
             (or (null newest)
                 (file-newer-than-file-p file newest)))
        (setq newest file)))
  newest))
```

Figure 1: Typical usage of `let` in Emacs Lisp.

```
(let ((file-name-handler-alist nil)
      (format-alist nil)
      (after-insert-file-functions nil)
      (coding-system-for-read 'binary)
      (coding-system-for-write 'binary)
      (find-buffer-file-type-function
       (if (fboundp 'find-buffer-file-type)
           (symbol-function 'find-buffer-file-type)
         nil)))
  (unwind-protect
      (progn
        (fset 'find-buffer-file-type
              (lambda (filename) t))
        (insert-file-contents
          filename visit start end replace))
    (if find-buffer-file-type-function
        (fset 'find-buffer-file-type
              find-buffer-file-type-function)
      (fmakunbound 'find-buffer-file-type))))
```

Figure 2: Parameterizations via dynamic `let` in Emacs Lisp.

flow analysis. This paper presents such an analysis called *dynamic scope analysis.*

Specifically, our contributions are the following:

- We have formulated a semantics for a subset of Emacs Lisp, called Mini Emacs Lisp, similar to the *sequential evaluation function* for Λ_d by Moreau [20].
- We have applied the flow analysis framework of Nielson and Nielson [22] to the semantics, resulting in an acceptability relation for flow analyses of Mini Emacs Lisp programs.
- We have used the acceptability relation to formulate and implement a flow analysis for Emacs Lisp which tracks the flow of bindings in addition to the flow of values.
- We have applied the analysis to real Emacs Lisp code. More specifically, the analysis is able to handle medium-sized real-world examples with high accuracy and reasonable efficiency.

The work presented here is a part of the `el2scm` project that works on the migration from Emacs Lisp to Scheme. However, the other aspects of the translation (such as front-end issues, correct handling of symbols, the code-data duality, treatment of primitives and so on) are outside the (lexical) scope of this paper. Indeed, the analysis could be used for a number of other purposes, among them the development of an efficient compiler for Emacs Lisp, or the translation to a different substrate such as Common Lisp.

Overview. The next section presents some code examples which show the need for a dynamic scope analysis. Section 3 defines the syntax of Mini Emacs Lisp. Section 4 develops an operational semantics with evaluation contexts. Based on the semantics, Section 5 presents a specification of a correct flow analysis. The next section sketches a correctness proof. Our implementation approach is described in Section 7. Section 8 describes some experimental results gained with our implementation prototype. We end with a discussion of related work and a conclusion.

2. EXAMPLES

Consider the Emacs Lisp code shown in Figure 1, taken literally from `files.el` in the current XEmacs core. It contains five variable bindings, all introducing temporary names for intermediate values. The bindings of the variables `filename`, `file`, `dir`, `comp`, and `newest` are all visible in the other functions reachable from the body of the `let`, yet none of them contain occurrences of these names. The only variable occurrences which access the bindings are in the body of the `let*` itself, and all are within the lexical scope of the bindings. Hence, translating the `let*` into a lexically-scoped counterpart in the target language would preserve the behavior of this function.

Figure 2 shows an example for idiomatic use of dynamic binding (also taken from `files.el`): It is part of the implementation of `insert-file-contents-literally` which calls `insert-file-contents` in the body of the `let`. The definition of `insert-file-contents` indeed contains occurrences of the variables bound in the `let` with the exception of `find-buffer-file-type-function`. Therefore, it is not permissible to translate the `let` with a lexically-scoped binding construct.

For the vast majority of binding constructs in real Emacs Lisp code, dynamic scope and lexical scope coincide. Thus, the ultimate goal of the analysis is to detect as many of these bindings constructs as possible.

In general however, value flow and the flow of bindings interact during the evaluation of Emacs Lisp programs. Hence, it is not possible to apply standard flow analyses based on lexical-binding semantics to solve the problem; a new analysis is necessary.

3. SYNTAX OF MINI EMACS LISP

For the sake of simplicity, we concentrate on a subset of Emacs Lisp called *Mini Emacs Lisp* in the paper. We omit multi-parameter (and variable-parameter) functions, `catch/throw`, dual name spaces for functions and "ordinary" values, the resulting gratuitous split between `funcall` and regular application as well as the data/code duality which appears in various contexts in Emacs Lisp. Adding these features to the analysis is straightforward and does not require significant new insights, which is why we omit it here. Our implementation of the analysis does treat all of these features.

Here is the syntax for Mini Emacs Lisp:

$$
\begin{array}{rcll}
l & \in & \textbf{Lab} & ::= \ \ldots \\
s, x & \in & \textbf{SymVar} & ::= \ \texttt{fritz} \mid \texttt{franz} \mid \ldots \\
c & \in & \textbf{Lit} & ::= \ \texttt{0} \mid \texttt{1} \mid \texttt{2} \mid \ldots \\
b & \in & \textbf{Prim} & ::= \ \texttt{cons} \mid \texttt{car} \mid \ldots \\
\\
t & \in & \textbf{Term} & ::= \ c \\
 & & & \quad \mid \ \texttt{(quote } s\texttt{)} \\
 & & & \quad \mid \ \texttt{(lambda (}x\texttt{) } e\texttt{)} \\
 & & & \quad \mid \ x \\
 & & & \quad \mid \ \texttt{(setq } x\ e\texttt{)} \\
 & & & \quad \mid \ \texttt{(}e_0\ e_1\texttt{)} \\
 & & & \quad \mid \ \texttt{(let } x\ e_1\ e_2\texttt{)} \\
 & & & \quad \mid \ \texttt{(if } e_0\ e_1\ e_2\texttt{)} \\
 & & & \quad \mid \ \texttt{(}b\ e_1 \ldots e_n\texttt{)} \\
\\
e & \in & \textbf{Exp} & ::= \ t^l \\
\\
d & \in & \textbf{Def} & ::= \ \texttt{(defvar } x\ e\texttt{)} \\
 & & & \quad \mid \ \texttt{(defun } x_0\ \texttt{(}x_1\texttt{) } e\texttt{)} \\
\\
p & \in & \textbf{Prg} & ::= \ d^*\ e
\end{array}
$$

All expressions carry unique labels which the analysis uses for identifying locations in the program source. The set of literals is trivially extensible. Note that Emacs Lisp uses the `nil` symbol for boolean false, and everything else for true.

An Emacs Lisp program consists of a sequence of definitions followed by a single expression—the entry point of the program.

4. A SEMANTICS FOR MINI EMACS LISP

We present a structural operational or small-step semantics [23] for Mini Emacs Lisp. We use evaluation contexts and syntactic rewriting as developed by Felleisen and Friedman [6].

4.1 Values and Intermediate Terms

We use separate syntactic categories for intermediate expressions and values. Here is the syntax for literals and abstractions:

$$
\begin{array}{rcll}
f & \in & \textbf{Fun} & ::= \ \texttt{(func (}x\texttt{) } e\texttt{)} \\
\\
v & \in & \textbf{Val} & ::= \ \texttt{(prim } c\texttt{)} \\
 & & & \quad \mid \ \texttt{(sym } s\texttt{)} \\
 & & & \quad \mid \ f \\
 & & & \quad \mid \ \texttt{(pair } v_1^{l_1}\ v_2^{l_2}\texttt{)} \\
\\
it & \in & \textbf{ITerm} & ::= \ \texttt{(bind } x\ v\ e\texttt{)} \\
\\
e & \in & \textbf{Exp} & ::= \ v^l \mid it^l
\end{array}
$$

The elements of **Val**, called *values*, are results from successful computations. They represent primitive values, symbols and functions, and correspond to the the expressions of **Exp** which produce them.

The semantics uses intermediate `bind` terms to handle dynamic binding: They result from reducing `let` expressions with the value to be bound to the variable already evaluated. Expressions attach labels to values and intermediate terms. Only the value bound to a variable by a `bind` term does not carry a label because `bind` expressions only show up during evaluation, but not in the analysis which only looks at the source code.

4.2 Environments

Environments ρ are finite mapping from symbols to values and contain bindings:

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

The notation for 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 $\rho[x \mapsto v]$.

4.3 Evaluation Contexts

Here are the evaluation contexts for Mini Emacs Lisp:

$$
\begin{array}{rcll}
\mathcal{E} & \in & \textbf{EvalContext} & ::= \ [-] \\
 & & & \quad \mid \ \texttt{(}\mathcal{E}\ e_2\texttt{)}^l \\
 & & & \quad \mid \ \texttt{(if } \mathcal{E}\ e_1\ e_2\texttt{)}^l \\
 & & & \quad \mid \ \texttt{(let } x\ \mathcal{E}\ e_2\texttt{)}^l \\
 & & & \quad \mid \ \texttt{(bind } x\ v\ \mathcal{E}\texttt{)}^l \\
 & & & \quad \mid \ \texttt{(setq } x\ \mathcal{E}\texttt{)}^l \\
 & & & \quad \mid \ \texttt{(}p\ v^*\ \mathcal{E}\ e^*\texttt{)}^l \\
\\
\mathcal{V}_{x_0} & \in & \textbf{VarContext}(x_0) & ::= \ [-] \\
 & & & \quad \mid \ \texttt{(}\mathcal{E}\ e_2\texttt{)}^l \\
 & & & \quad \mid \ \texttt{(if } \mathcal{V}_{x_0}\ e_1\ e_2\texttt{)}^l \\
 & & & \quad \mid \ \texttt{(let } x\ \mathcal{V}_{x_0}\ e_2\texttt{)}^l \\
 & & & \quad \mid \ \texttt{(bind } x_1\ v\ \mathcal{V}_{x_0}\texttt{)}^l \\
 & & & \qquad\qquad \text{if } x_0 \neq x_1 \\
 & & & \quad \mid \ \texttt{(setq } x\ \mathcal{V}_{x_0}\texttt{)}^l \\
 & & & \quad \mid \ \texttt{(}b\ v^*\ \mathcal{V}_{x_0}\ e^*\texttt{)}^l
\end{array}
$$

The rules for **EvalContext** describes all contexts in which a reduction step in Mini Emacs Lisp can occur. Variable access needs the most recent dynamic binding of the variable. The *variable contexts* in **VarContext**(x_0) help accomplish this; they describe all contexts that do not contain any bindings associated with the symbol x_0.

4.4 Reductions

An evaluation state consists of a partially evaluated expression and a global environment. Thus, a *configuration* γ of **Conf** is a tuple consisting of an environment and a current expression:

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

The primitive steps of the evaluation process are reduction rules. Some expressions immediately reduce to a value:

$$
\begin{array}{ll}
[\textbf{c}] & \rho, \mathcal{E}[c^l] \rightarrow \rho, \mathcal{E}[\texttt{(prim } c\texttt{)}^l] \\
\\
[\textbf{quote}] & \rho, \mathcal{E}[\texttt{(quote } s\texttt{)}^l] \rightarrow \rho, \mathcal{E}[\texttt{(sym } s\texttt{)}^l] \\
\\
[\textbf{lambda}] & \rho, \mathcal{E}[\texttt{(lambda (}x\texttt{) } e\texttt{)}^l] \rightarrow \rho, \mathcal{E}[\texttt{(func (}x\texttt{) } e\texttt{)}^l]
\end{array}
$$

Note that in Emacs Lisp, abstractions do not evaluate to closures—this is dynamic scope, after all.

Here are the semantic mechanics for dealing with variable access:

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

$$[\textbf{var}_{\textbf{glob}}] \quad \rho, \mathcal{V}_x[x^l] \rightarrow \rho, \mathcal{V}_x[v^l] \quad \text{if } x \in \mathit{dom}(\rho),\ \rho(x) = v$$

A variable may have either a local or a global binding. The `let` and `lambda` constructs introduce local bindings. For a variable occurrence, the closest `bind` context for that variable holds its value. The [**var**] rule expresses this behavior; the context $\mathcal{V}_x$ guarantees that there is no other binding closer to the variable. Lacking a local binding, a global one must apply; the [$\textbf{var}_{\textbf{glob}}$] rule takes over.

The machinery for mutating bindings by `setq` is analogous to the one for referencing variables:

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

$$[\textbf{setq}_{\textbf{global}}] \quad \rho, \mathcal{V}_x[\texttt{(setq } x \ v_0^{l_0}\texttt{)}^l] \rightarrow \rho[x \mapsto v_0], \mathcal{V}_x[v_0^l]$$

In the case of a local binding, the [**setq**] rule changes the value in the corresponding `bind` context. Assignments to global variables mutate the global environment.

Here are the reductions for function applications and local variable bindings:

$$[\textbf{app}] \quad \rho, \mathcal{E}[\texttt{((func (}x_1\texttt{) } e_2\texttt{)}^{l_0} \ e_1\texttt{)}^l] \rightarrow \rho, \mathcal{E}[\texttt{(let } x_1 \ e_1 \ e_2\texttt{)}^l]$$

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

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

The [**app**] rule reduces a function application to a binding of the function parameter wrapped around the function body and the environment.

The [**let**] rule of **EvalContext** turns a `let` expression into a corresponding `bind` expression. Evaluation continues with the body e_2 until it becomes a value. Then, the [**bind**] rule removes the obsolete context.

Note that the distinction between `let` expressions and `bind` expressions is unnecessary when considering only the semantics, but the formulation of the flow analysis requires their separation.

The [$\textbf{if}_1$] and [$\textbf{if}_2$] rules handle conditionals:

$$[\textbf{if}_1] \quad \rho, \mathcal{E}[\texttt{(if } v^{l_0} \ t_1^{l_1} \ e_2\texttt{)}^l] \rightarrow \rho, \mathcal{E}[t_1^l] \quad \text{if } v \neq \texttt{(sym nil)}$$

$$[\textbf{if}_2] \quad \rho, \mathcal{E}[\texttt{(if } v^{l_0} \ e_1 \ t_2^{l_2}\texttt{)}^l] \rightarrow \rho, \mathcal{E}[t_2^l] \quad \text{if } v = \texttt{(sym nil)}$$

Here are reduction rules for selected primitives, namely those dealing with pairs:

$$[\textbf{cons}] \quad \rho, \mathcal{E}[\texttt{(cons } v_1^{l_1} \ v_2^{l_2}\texttt{)}^l] \rightarrow \rho, \mathcal{E}[\texttt{(pair } v_1^{l_1} \ v_2^{l_2}\texttt{)}^l]$$

$$[\textbf{car}] \quad \rho, \mathcal{E}[\texttt{(car (pair } v_1^{l_1} \ v_2^{l_2}\texttt{)}^{l_0}\texttt{)}^l] \rightarrow \rho, \mathcal{E}[v_1^l]$$

$$[\textbf{cdr}] \quad \rho, \mathcal{E}[\texttt{(cdr (pair } v_1^{l_1} \ v_2^{l_2}\texttt{)}^{l_0}\texttt{)}^l] \rightarrow \rho, \mathcal{E}[v_2^l]$$

The [**cons**] rule produces a `pair` value from two argument values. `Car` selects the first component of pairs by rule [**car**], the [**cdr**] rule handles `cdr`.

The combination of the reduction rules defines the reduction relation

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

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

4.5 Expression Contexts

So far, only the meaning of expression is defined by the $\rightarrow$ relation. For programs, we define another kind of context, the *expression contexts* of **ExpContext**:

$$\begin{array}{lcll} \mathcal{X} & \in & \textbf{ExpContext} ::= & \texttt{(defvar } x \ [-]\texttt{)} \ p \\ & & \quad\quad\quad\quad\ \mid & [-] \end{array}$$

4.6 Reductions for Programs

Equipped with the notion of *program configurations*

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

as well as contexts for programs $\mathcal{X}$ and the reduction relation $\rightarrow$ for expressions, it is possible to state the rewriting rules $\rightarrow_d$ for programs:

$$[\textbf{defvar}] \quad \rho, \texttt{(defvar } x \ v_0^{l_0}\texttt{)} \ p \rightarrow_d \rho[x \mapsto v_0], p \quad \text{if } x \notin \mathit{dom}(\rho)$$

$$[\textbf{defun}] \quad \rho, \texttt{(defun } x_0 \ \texttt{(}x_1\texttt{)} \ e\texttt{)} \ p \rightarrow_d \rho[x_0 \mapsto \texttt{(func (}x_1\texttt{) } e\texttt{)}], p \quad \text{if } x_0 \notin \mathit{dom}(\rho)$$

$$[\textbf{exp}] \quad \frac{\rho, e \rightarrow \rho', e'}{\rho, \mathcal{X}[e] \rightarrow_d \rho', \mathcal{X}[e']}$$

The [**defvar**] and [**defun**] rules satisfy top-level definitions. The [**defvar**] rule inserts the new global binding in the variable environment ρ. The condition $x \notin \mathit{dom}(\rho)$ guarantees that there is only one global variable for every name. The [**defun**] rule does the equivalent for procedures. The [**exp**] rule allows the use of all the reductions for expressions at the places defined using the contexts **ExpContext**. Again $\rightarrow_d{}^*$ is the reflexive transitive closure of $\rightarrow_d$.

4.7 The Evaluation Function of Programs

The reduction relation $\rightarrow_d$ rewrites the program until it gets a final answer. This does not always happen: the program may loop in which case the reduction sequence is infinite, or evaluation may get stuck at a configuration with no matching reduction rule. Thus, the reduction relation induces a partial evaluation function **eval**:

$$\textbf{eval} : \textbf{Prg} \dashrightarrow \textbf{Val}$$

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

5. SPECIFICATION OF THE ANALYSIS

This section specifies a flow analysis for Mini Emacs Lisp. With the help of the definitions for the abstract domains of the analysis we define an *acceptability relation* for correct flow analyses which employ these domains. The actual analysis results directly from the definition of the acceptability relation.

5.1 Abstract Domains

Here are the abstract domains of the analysis:

$$\begin{array}{rcll} bp & \in & \widehat{\mathbf{BP}} & = \mathbf{Lab} \cup \{\diamond\} \\ bpe & \in & \widehat{\mathbf{BPEnv}} & = \mathbf{SymVar} \to \widehat{\mathbf{BP}} \\ \\ \hat{\mathsf{p}} & \in & \widehat{\mathbf{Cons}} & = \mathbf{Lab} \times \mathbf{Lab} \times \widehat{\mathbf{BPEnv}} \\ \hat{\mathsf{v}} & \in & \widehat{\mathbf{Val}} & = \mathcal{P}(\mathbf{SymVar} \cup \{\omega\} \cup \mathbf{Fun} \cup \widehat{\mathbf{Cons}}) \\ \\ \hat{\rho} & \in & \widehat{\mathbf{Env}} & = (\mathbf{SymVar} \times \widehat{\mathbf{BP}}) \to \widehat{\mathbf{Val}} \\ \hat{\mathsf{C}} & \in & \widehat{\mathbf{Cache}} & = (\mathbf{Lab} \times \widehat{\mathbf{BPEnv}}) \to \widehat{\mathbf{Val}} \end{array}$$

Birthplaces—$\widehat{\mathbf{BP}}$ for short—denote syntactic locations of variable bindings. The $\diamond$ stands for top-level bindings. The label of the body of a function or of a `let` expression serves as the birthplace for the binding it creates.

Birthplace environments $\widehat{\mathbf{BPEnv}}$ are abstractions over regular variable environments; they map variables to birthplaces instead of regular values.

$\widehat{\mathbf{Cons}}$ 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. The two labels are the labels of these two argument subexpressions of the `cons` expression which created the pair. The birthplace environment registers the abstract bindings active at the time of creation of the pair. Registering the birthplace environment is necessary because we differentiate program points depending on the birthplace environments they occur under.

$\widehat{\mathbf{Val}}$ is the set of all possible *abstract values* $\hat{\mathsf{v}}$. An abstract value represents a set of run-time values. Not every run-time value is relevant to the analysis: the single symbol ω represents all primitive values except for symbols. The analysis tracks symbols needed (eventually) for variable names, functions, primitive values, and abstract pairs.

An *abstract cache* $\hat{\mathsf{C}}$ of $\widehat{\mathbf{Cache}}$ is an abstract profile of all values which occur during a program run. It tracks the abstract values of program subexpressions, differentiated by birthplace environment.

An *abstract environment* $\hat{\rho}$ of $\widehat{\mathbf{Env}}$ is a union of the environments that occur during the evaluation of a program. It associates a variable name and one of its birthplaces with an abstract value.

5.2 Acceptability for Programs

We define an *acceptability relation for programs* $\models$:

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

The $\models$ relation defines the validity of analyses $(\hat{\mathsf{C}}, \hat{\rho})$ with regard to a program p and a current birthplace environment bpe. From now on, the notation is

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

5.2.1 Value Expressions

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

The [**c**], [**quote**], and [**lam**] clauses register their abstract counterpart in the abstract cache under the program point l and the current birthplace environment bpe. Note that [**lam**] does not require that the analysis is also valid for the body of each `lambda` term, because an acceptable analysis must only treat the *reachable* functions correctly.

5.2.2 Expressions

Occurrences of variable references and mutations induce further validity constraints. The [**var**] rule for variable references enforces that the abstract value for the variable x and its current birthplace $bpe(x)$, held in the abstract environment, must be a subset of the abstract value that linked it to its label and birthplace environment in the abstract cache:

$$\begin{array}{ll} [\mathbf{var}] & (\hat{\mathsf{C}}, \hat{\rho}) \models_{bpe} x^l \\ & \mathbf{iff} \quad \hat{\rho}(x, bpe(x)) \subseteq \hat{\mathsf{C}}(l, bpe) \end{array}$$

The [**setq**] clause enforces that the analysis for the right-hand side is also valid. Moreover, a valid analysis allows values that result from the subexpression t_0 to be possible values for the variable x under the current bindings bpe and also for the whole expression:

$$\begin{array}{ll} [\mathbf{setq}] & (\hat{\mathsf{C}}, \hat{\rho}) \models_{bpe} \texttt{(setq } x\ t_0^{l_0}\texttt{)}^l \\ & \mathbf{iff}\ (\hat{\mathsf{C}}, \hat{\rho}) \models_{bpe} t_0^{l_0} \wedge \\ & \qquad \hat{\mathsf{C}}(l_0, bpe) \subseteq \hat{\rho}(x, bpe(x)) \wedge \\ & \qquad \hat{\mathsf{C}}(l_0, bpe) \subseteq \hat{\mathsf{C}}(l, bpe) \end{array}$$

The [**app**] clause specifies the constraints for procedure calls. Its first and second condition, $(\hat{\mathsf{C}}, \hat{\rho}) \models_{bpe} t_0^{l_0}$ and $(\hat{\mathsf{C}}, \hat{\rho}) \models_{bpe} t_1^{l_1}$, guarantee that the analysis is also valid under the same birthplace environment for the operator t_0 and the operand t_1:

$$\begin{array}{ll} [\mathbf{app}] & (\hat{\mathsf{C}}, \hat{\rho}) \models_{bpe} \texttt{(}t_0^{l_0}\ t_1^{l_1}\texttt{)}^l \\ & \mathbf{iff}\ (\hat{\mathsf{C}}, \hat{\rho}) \models_{bpe} t_0^{l_0} \wedge \\ & \quad (\hat{\mathsf{C}}, \hat{\rho}) \models_{bpe} t_1^{l_1} \wedge \\ & \quad (\forall \texttt{(func (}x_1\texttt{) } t_b^{l_b}\texttt{)} \in \hat{\mathsf{C}}(l_0, bpe). \\ & \qquad (\hat{\mathsf{C}}, \hat{\rho}) \models_{bpe_0} t_b^{l_b} \wedge \\ & \qquad \hat{\mathsf{C}}(l_1, bpe) \subseteq \hat{\rho}(x_1, l_b) \wedge \\ & \qquad \hat{\mathsf{C}}(l_b, bpe_0) \subseteq \hat{\mathsf{C}}(l, bpe) \wedge \\ & \qquad \mathbf{where} \quad bpe_0 = bpe[x_1 \mapsto l_b]) \end{array}$$

Every function $\texttt{(func (}x_1\texttt{) } t_b^{l_b}\texttt{)}$ which might occur in the operator position t_0 of a procedure call under bpe must have a valid analysis for its body as well, under an expanded birthplace environment bpe_0 which contains a binding for the function parameter x_1. Moreover, the analysis must link the abstract values of the argument with those of the formal parameter x_1 as well as the possible results of the body with the those of the whole expression.

The [**let**] clause works analogously to function application:

$$\begin{array}{ll} [\mathbf{let}] & (\hat{\mathsf{C}}, \hat{\rho}) \models_{bpe} \texttt{(let } x\ t_1^{l_1}\ t_2^{l_2}\texttt{)}^l \\ & \mathbf{iff}\ (\hat{\mathsf{C}}, \hat{\rho}) \models_{bpe} t_1^{l_1} \wedge \\ & \quad (\hat{\mathsf{C}}, \hat{\rho}) \models_{bpe_0} t_2^{l_2} \wedge \\ & \quad \hat{\mathsf{C}}(l_1, bpe) \subseteq \hat{\rho}(x, l_2) \wedge \\ & \quad \hat{\mathsf{C}}(l_2, bpe_0) \subseteq \hat{\mathsf{C}}(l, bpe) \\ & \quad \mathbf{where} \quad bpe_0 = bpe[x \mapsto l_2] \end{array}$$

In the [**if**] clause, each branch contributes to a valid analysis:

$$\begin{array}{ll} [\textbf{if}] & (\hat{\mathsf{C}},\hat{\rho}) \models_{bpe} \texttt{(if } t_0^{l_0}\ t_1^{l_1}\ t_2^{l_2}\texttt{)}^l \\ & \textbf{iff } (\hat{\mathsf{C}},\hat{\rho}) \models_{bpe} t_0^{l_0} \wedge \\ & \quad (\hat{\mathsf{C}},\hat{\rho}) \models_{bpe} t_1^{l_1} \wedge \\ & \quad \hat{\mathsf{C}}(l_1, bpe) \subseteq \hat{\mathsf{C}}(l, bpe) \wedge \\ & \quad (\hat{\mathsf{C}},\hat{\rho}) \models_{bpe} t_2^{l_2} \wedge \\ & \quad \hat{\mathsf{C}}(l_2, bpe) \subseteq \hat{\mathsf{C}}(l, bpe) \end{array}$$

5.2.3 *Primitives*

Each primitive has its own associated flow behavior. Pair construction and selection serve as examples.

The [**cons**] rule for the pair constructor is straightforward: A `cons` produces an abstract pair from abstract values with the labels of its arguments, under the original birthplace environment:

$$\begin{array}{ll} [\textbf{cons}] & (\hat{\mathsf{C}},\hat{\rho}) \models_{bpe} \texttt{(cons } t_1^{l_1}\ t_2^{l_2}\texttt{)}^l \\ & \textbf{iff } (\hat{\mathsf{C}},\hat{\rho}) \models_{bpe} t_1^{l_1} \wedge \\ & \quad (\hat{\mathsf{C}},\hat{\rho}) \models_{bpe} t_2^{l_2} \wedge \\ & \quad (l_1, l_2, bpe) \in \hat{\mathsf{C}}(l, bpe) \end{array}$$

The [**car**] and [**cdr**] clauses are also straightforward: They induce validity constraints on the argument, and then simply pick the first or second component respectively of the abstract pairs flowing into it:

$$\begin{array}{ll} [\textbf{car}] & (\hat{\mathsf{C}},\hat{\rho}) \models_{bpe} \texttt{(car } t_0^{l_0}\texttt{)}^l \\ & \textbf{iff } (\hat{\mathsf{C}},\hat{\rho}) \models_{bpe} t_0^{l_0} \wedge \\ & \quad (\forall (l_1, l_2, bpe_0) \in \hat{\mathsf{C}}(l_0, bpe). \\ & \quad\quad \hat{\mathsf{C}}(l_1, bpe_0) \subseteq \hat{\mathsf{C}}(l, bpe)) \\ [\textbf{cdr}] & (\hat{\mathsf{C}},\hat{\rho}) \models_{bpe} \texttt{(cdr } t_0^{l_0}\texttt{)}^l \\ & \textbf{iff } (\hat{\mathsf{C}},\hat{\rho}) \models_{bpe} t_0^{l_0} \wedge \\ & \quad (\forall (l_1, l_2, bpe_0) \in \hat{\mathsf{C}}(l_0, bpe). \\ & \quad\quad \hat{\mathsf{C}}(l_2, bpe_0) \subseteq \hat{\mathsf{C}}(l, bpe)) \end{array}$$

5.2.4 *Definitions*

The [**defvar**] and [**defun**] clauses extend the notion of valid scope analyses to entire programs. The [**defvar**] clause handles variable definitions:

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

A valid analysis must reflect the value initially bound to the variable. It must also associate the variable x with its abstract values under the current binding $bpe(x)$. Moreover, a valid analysis must take into account the rest of the program p, too.

The [**defun**] clause registers the procedure in the abstract environment. As in the [**defvar**] clause, the rest of the program p must be valid as well.

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

5.2.5 *Values*

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. Now, the next goal is to show that valid analyses agree with the semantics—that is, that they are semantically correct. However, the reduction rules generate intermediate terms not covered by the rules so far. Here they are:

$$\begin{array}{ll} [\textbf{const}] & (\hat{\mathsf{C}},\hat{\rho}) \models_{bpe} \texttt{(prim } c\texttt{)}^l \\ & \textbf{iff } \omega \in \hat{\mathsf{C}}(l, bpe) \\ [\textbf{sym}] & (\hat{\mathsf{C}},\hat{\rho}) \models_{bpe} \texttt{(sym } s\texttt{)}^l \\ & \textbf{iff } s \in \hat{\mathsf{C}}(l, bpe) \\ [\textbf{proc}] & (\hat{\mathsf{C}},\hat{\rho}) \models_{bpe} \texttt{(func (}x\texttt{)}\ e_0\texttt{)}^l \\ & \textbf{iff } \texttt{(func (}x\texttt{)}\ e_0\texttt{)} \in \hat{\mathsf{C}}(l, bpe) \end{array}$$

The [**const**], [**sym**], and [**proc**] clauses are identical to their equivalents [**c**], [**quote**], and [**lam**] because their semantics are identical.

The [**pair**] clause is a simpler version of the [**cons**] clause. The only difference is—since a pair already carries two values in it—that it is unknown under which prior birthplace environment the evaluation took place. The only requirement is that a suitable birthplace environment bpe_0 exists:

$$\begin{array}{ll} [\textbf{pair}] & (\hat{\mathsf{C}},\hat{\rho}) \models_{bpe} \texttt{(pair } v_1^{l_1}\ v_2^{l_2}\texttt{)}^l \\ & \textbf{iff } \exists bpe_0 . (\hat{\mathsf{C}},\hat{\rho}) \models_{bpe_0} v_1^{l_1} \wedge \\ & \quad (\hat{\mathsf{C}},\hat{\rho}) \models_{bpe_0} v_2^{l_2} \wedge \\ & \quad (l_1, l_2, bpe_0) \in \hat{\mathsf{C}}(l, bpe) \end{array}$$

5.2.6 *Intermediate Expressions*

The final clause in the definition of acceptability handles intermediate `bind` expressions. A `bind` expression binds a variable x to a value v_1 during the evaluation of the body t_2:

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

The [**bind**] rule requires a valid analysis for the body under a suitably extended birthplace environment bpe_0. Moreover, the value of the body becomes the value of the `bind` expression. The supplementary constraint $v_1\, \mathcal{A}\, (\hat{\rho}(x, l_2), \hat{\mathsf{C}})$ reflects that the actual new binding also has to show up in the abstract variable environment under the the relevant birthplace l_2; the $\mathcal{A}$ relation is explained in the next section.

5.3 The Approximation Relation $\mathcal{A}$

Intuitively, the $\models$ relation determines if a dynamic scope analysis $(\hat{\mathsf{C}},\hat{\rho})$ correctly reflects the evaluation process of a program a in an abstract sense. The formulation of $\models$ uses the *approximation relation* $\mathcal{A}$ that regulates the approximation of values **Val** with abstract equivalents. Here is its formal definition:

$$\begin{array}{lll} \multicolumn{3}{l}{\mathcal{A} \subseteq \mathbf{Val} \times \widehat{\mathbf{Val}} \times \widehat{\mathbf{Cache}}} \\ \multicolumn{3}{l}{v\, \mathcal{A}\, (\hat{\mathsf{v}}, \hat{\mathsf{C}})} \\ \textbf{iff } \forall c \forall s \forall f \forall v_1 \forall v_2 . & \multicolumn{2}{l}{((v = c \Rightarrow \omega \in \hat{\mathsf{v}}) \wedge} \\ & \multicolumn{2}{l}{(v = s \Rightarrow s \in \hat{\mathsf{v}}) \wedge} \\ & \multicolumn{2}{l}{(v = f \Rightarrow f \in \hat{\mathsf{v}}) \wedge} \\ & \multicolumn{2}{l}{(v = \texttt{(pair } v_1^{l_1}\ v_2^{l_2}\texttt{)} \Rightarrow} \\ & \quad \exists bpe . & (l_1, l_2, bpe) \in \hat{\mathsf{v}} \wedge \\ & & v_1\, \mathcal{A}\, (\hat{\mathsf{C}}(l_1, bpe), \hat{\mathsf{C}}) \wedge \\ & & v_2\, \mathcal{A}\, (\hat{\mathsf{C}}(l_2, bpe), \hat{\mathsf{C}}))) \end{array}$$

$\mathcal{A}$ holds between a value v and its correct representation as a set of abstract values and an abstract cache. This is straightforward except for the treatment of pairs: The representation of a pair consists of its components' creation points and a birthplace environment. An abstract representation however must also map to abstract *values* for its components. This is why a value cache $\hat{\mathsf{C}}$ participates in the definition of $\mathcal{A}$.

5.4 The Well-Definedness of $\models$

It is not immediately clear that the acceptability relation $\models$ from Subsection 5.2 is unambiguous. Structural induction by itself is not sufficient because the [**app**] clause is not compositional.

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

$$\mathcal{Q} : \mathcal{P}(\widehat{\mathbf{Cache}} \times \widehat{\mathbf{Env}} \times \widehat{\mathbf{BPEnv}} \times \mathbf{Exp}) \rightarrow \mathcal{P}(\widehat{\mathbf{Cache}} \times \widehat{\mathbf{Env}} \times \widehat{\mathbf{BPEnv}} \times \mathbf{Exp})$$

with

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

This change in perspective leads to a specification of $\models$ using sound mathematical means. $\mathcal{Q}$ is a monotone function on the complete lattice

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

because

- $(\mathcal{P}(\widehat{\mathbf{Cache}} \times \widehat{\mathbf{Env}} \times \widehat{\mathbf{BPEnv}} \times \mathbf{Exp}), \sqsubseteq)$ 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, $\forall R_1, R_2. R_1 \sqsubseteq R_2 \Rightarrow \mathcal{Q}(R_1) \sqsubseteq \mathcal{Q}(R_2)$.

Consequently, $\mathcal{Q}$ has a greatest fixed point. Thus, a well-defined definition of $\models$ works by coinduction as

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

5.5 Acceptability for Environments

Since dynamic scope analysis is ultimately concerned with scope and hence with environments, it is necessary to extend the notion of acceptability to environments:

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

$$(\hat{\mathsf{C}}, \hat{\rho}) \models_{bpe} \rho \quad \textbf{iff} \quad \forall x \in \mathit{dom}(\rho). \rho(x)\, \mathcal{A}\, (\hat{\rho}(x, bpe(x)), \hat{\mathsf{C}})$$

This acceptability relation for environments examines every binding in an actual environment which occurs during evaluation and relates it to its abstract counterpart for correctness.

5.6 Acceptability for Configurations

The combination of the acceptability relation for programs with that for environments produces an acceptability relation for *configurations*—combinations of environments and expressions:

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

$$\begin{array}{lll}(\hat{\mathsf{C}}, \hat{\rho}) \models_{bpe} \rho, e & \textbf{iff} & (\hat{\mathsf{C}}, \hat{\rho}) \models_{bpe} \rho \wedge\\ & & (\hat{\mathsf{C}}, \hat{\rho}) \models_{bpe} e\end{array}$$

Furthermore, it is possible to define an acceptability relation for *program configurations*—combinations of programs and environments:

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

$$\begin{array}{lll}(\hat{\mathsf{C}}, \hat{\rho}) \models_{bpe} \rho, p & \textbf{iff} & (\hat{\mathsf{C}}, \hat{\rho}) \models_{bpe} \rho \wedge\\ & & (\hat{\mathsf{C}}, \hat{\rho}) \models_{bpe} p\end{array}$$

6. SEMANTIC CORRECTNESS

The semantics developed in Section 4 employs evaluation contexts and rewriting rules. Hence, the specification of the semantics uses almost exclusively syntactical means with the exception of the notion of environments: a program transitions through a sequence of configurations which include valid programs or expressions until it reaches a final value, gets stuck or loops forever.

The definition of the acceptability relation in the previous section was derived intuitively. A correctness proof is necessary which must show that every valid analysis stays valid under the evaluation process.

This section summarizes the most import lemmas and theorems involved in the proof. For details, the reader is referred to Neubauer's thesis dissertation [21]. The first lemma states 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 $(\hat{\mathsf{C}}, \hat{\rho}) \models_{bpe} v^l$ *iff* $v\, \mathcal{A}\, (\hat{\mathsf{C}}(l, bpe), \hat{\mathsf{C}})$

PROOF. By structural induction over v. □

Another lemma states 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* $v\, \mathcal{A}\, (\hat{\mathsf{v}}_1, \hat{\mathsf{C}})$ *and also* $\hat{\mathsf{v}}_1 \subseteq \hat{\mathsf{v}}_2$ *then* $v\, \mathcal{A}\, (\hat{\mathsf{v}}_2, \hat{\mathsf{C}})$.

PROOF. By structural induction over v. Each case of the proof is obtained individually by inspecting the definition of $\mathcal{A}$. □

The specification of the acceptability relation has the important property stated by the following lemma: if an analysis is valid for a term t at label t_1, and the abstract values flowing through it are all contained in the values flowing through label l_2, the analysis is also valid at label l_2:

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

PROOF. by case analysis over the rules of **Term**. As an example, here is the case for `setq` expressions:
From the first premise

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

follows

$$(\hat{\mathsf{C}}, \hat{\rho}) \models_{bpe} t_0^{l_0} \quad (1)$$

$$\hat{\mathsf{C}}(l_0, bpe) \subseteq \hat{\rho}(x, bpe(x)) \quad (2)$$

$$\hat{\mathsf{C}}(l_0, bpe) \subseteq \hat{\mathsf{C}}(l_1, bpe) \quad (3)$$

by the [**setq**] clause of $\models$. The assumption together with (3) yields

$$\hat{\mathsf{C}}(l_0, bpe) \subseteq \hat{\mathsf{C}}(l_2, bpe). \quad (4)$$

The backwards application of the [**setq**] clause together with (1) and (2) yields the proposition. The other cases work analogously. $\square$

Another central insight is that the validity of the dynamic scope analysis of an expression carries over those subexpressions which are in an evaluation context. Even stronger, such a subexpression can be replaced 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 [13]:

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

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

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

c) If also $\mathcal{E} \in \mathbf{VarContext}(x)$ then $bpe_0(x) = bpe(x)$.

PROOF. Structural induction over $\mathcal{E}$. $\square$

The first main theorem is subject reduction for expressions under the reduction relation $\rightarrow$. A valid dynamic scope analysis for an expression e and a correct approximation of the environment stay valid after one step with $\rightarrow$ for the resulting expression e' and the modified environment:

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

PROOF. By case analysis over the reduction relation $\rightarrow$. $\square$

The second theorem formulates subject reduction for entire programs, adapting the previous theorem one to the reduction relation $\rightarrow_p$:

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

PROOF. By case analysis over $\rightarrow_d$. $\square$

7. IMPLEMENTATION

The definition of the acceptability relation presented in Section 5.2 is a blueprint for a practical implementation of a dynamic scope analysis. Our own analysis is constraint-based [1]; it uses a set of syntactic entities to represent applications of the rules generated by the acceptability relation. The analysis, just like every other constraint-based program analysis, consists of two phases: constraint generation and constraint simplification.

In the following we consider a fixed program p_* and describe how to compute the least dynamic flow analysis for p_* which is acceptable with respect to the acceptability relation $\models$.

Since the program p_* is finite, it is possible to enumerate all its occurring labels, symbol, and functions. We call these finite sets $\mathbf{Lab}_*$, $\mathbf{SymVar}_*$ and $\mathbf{Fun}_*$, respectively. Similarly, the sets of possibly occurring birthplace environments $\widehat{\mathbf{BPEnv}}_*$ and possibly occurring abstract pairs $\widehat{\mathbf{Cons}}_*$ are also identifiable and finite. Accordingly, the finite set of all abstract values that are conceivable for all possible program runs of p_* is

$$a \in \mathbf{Abs}_* = \mathbf{SymVar}_* \cup \{\omega\} \cup \mathbf{Fun}_* \cup \widehat{\mathbf{Cons}}_*.$$

The finite sets serve as basis for the specification of the dynamic scope analysis for a program p_*.

7.1 Generating Constraints

In the constraints generated by the analysis, flow variables V stand for sets of abstract values. A flow variable $\mathsf{C}_{l,bpe}$ stands for the set of abstract values in the abstract cache at label l and birthplace environment bpe. A flow variable $\mathsf{r}_{x,l}$ stands for a set of abstract values in the abstract environment.

A constraint co in our analysis belongs to one of three different kinds. A *simple constraint* of the form

$$\{a\} \subseteq \mathsf{V},$$

where a is an abstract value of $\mathbf{Abs}_*$, states that a certain abstract value a must be member of the set of abstract values A. A *variable constraint*

$$\mathsf{V}_0 \subseteq \mathsf{V}_1$$

says that the abstract values of V_0 are all contained in those of V_1. A *conditional constraint*

$$\{a\} \subseteq \mathsf{V} \implies \mathsf{co}$$

where a is an abstract value of $\mathbf{Abs}_*$ and co is another constraint, states that the constraint co must hold if the abstract value a is a member of the set of abstract values denoted by V.

By inspecting the rules of the acceptability relation, we define the function $\mathcal{G}[\![p]\!]_{bpe\,\mathcal{M}}$ that constructs the set of constraints to be solved, as shown in Figure 3. Its first parameter is the program or expression for which constraints are generated. The second one, bpe, is the birthplace environment, relative to which the generation of the constraints takes place. The third parameter, $\mathcal{M}$, is a set of pairs of a label of a body of a procedure l_b and a birthplace environment bpe each. This set memoizes instances of pairs of procedures and birthplace environments already handled by the constraint generation. The analysis uses it to prevent generating duplicate constraints.

$$
\begin{array}{lcll}
\mathcal{G}\llbracket c^{l}\rrbracket_{bpe\,\mathcal{M}} & = & \{\{\omega\} \subseteq \mathsf{C}_{l,bpe}\} & \\
\mathcal{G}\llbracket \texttt{(quote } s\texttt{)}^{l}\rrbracket_{bpe\,\mathcal{M}} & = & \{\{s\} \subseteq \mathsf{C}_{l,bpe}\} & \\
\mathcal{G}\llbracket \texttt{(lambda (}x\texttt{) } e_0\texttt{)}^{l}\rrbracket_{bpe\,\mathcal{M}} & = & \{\{\texttt{(func (}x\texttt{) } e_0\texttt{)}\} \subseteq \mathsf{C}_{l,bpe}\} & \\
\mathcal{G}\llbracket x^{l}\rrbracket_{bpe\,\mathcal{M}} & = & \{\mathsf{r}_{x,bpe(x)} \subseteq \mathsf{C}_{l,bpe}\} & \\
\mathcal{G}\llbracket \texttt{(setq } x\ t_0^{l_0}\texttt{)}^{l}\rrbracket_{bpe\,\mathcal{M}} & = & \mathcal{G}\llbracket t_0^{l_0}\rrbracket_{bpe\,\mathcal{M}} & \cup\ \{\mathsf{C}_{l_0,bpe} \subseteq \mathsf{r}_{x,bpe(x)}\} \\
 & & & \cup\ \{\mathsf{C}_{l_0,bpe} \subseteq \mathsf{C}_{l,bpe}\} \\
\mathcal{G}\llbracket \texttt{(}t_0^{l_0}\ t_1^{l_1}\texttt{)}^{l}\rrbracket_{bpe\,\mathcal{M}} & = & \mathcal{G}\llbracket t_0^{l_0}\rrbracket_{bpe\,\mathcal{M}} & \cup\ \mathcal{G}\llbracket t_1^{l_1}\rrbracket_{bpe\,\mathcal{M}} \\
 & & & \cup\ \{\{\texttt{(func (}x_1\texttt{) } t_b^{l_b}\texttt{)}\} \subseteq \mathsf{C}_{l_0,bpe} \implies \mathsf{co} \\
 & & & \qquad \mid \texttt{(func (}x_1\texttt{) } t_b^{l_b}\texttt{)} \in \mathsf{Fun}_*, \\
 & & & \qquad bpe_0 = bpe[x_1 \mapsto l_b], \\
 & & & \qquad (l_b, bpe_0) \notin \mathcal{M}, \\
 & & & \qquad \mathcal{M}' = \mathcal{M} \cup \{(l_b, bpe_0)\}, \\
 & & & \qquad \mathsf{co} \in \mathcal{G}\llbracket t_b^{l_b}\rrbracket_{bpe_0\,\mathcal{M}'}\} \\
 & & & \cup\ \{\{\texttt{(func (}x_1\texttt{) } t_b^{l_b}\texttt{)}\} \subseteq \mathsf{C}_{l_0,bpe} \implies \mathsf{C}_{l_1,bpe} \subseteq \mathsf{r}_{x_1,l_b} \\
 & & & \qquad \mid \texttt{(func (}x_1\texttt{) } t_b^{l_b}\texttt{)} \in \mathsf{Fun}_*\} \\
 & & & \cup\ \{\{\texttt{(func (}x_1\texttt{) } t_b^{l_b}\texttt{)}\} \subseteq \mathsf{C}_{l_0,bpe} \implies \mathsf{C}_{l_b,bpe[x_1 1\mapsto l_b]} \subseteq \mathsf{C}_{l,bpe} \\
 & & & \qquad \mid \texttt{(func (}x_1\texttt{) } t_b^{l_b}\texttt{)} \in \mathsf{Fun}_*\} \\
\mathcal{G}\llbracket \texttt{(let } x\ t_1^{l_1}\ t_2^{l_2}\texttt{)}^{l}\rrbracket_{bpe\,\mathcal{M}} & = & \mathcal{G}\llbracket t_1^{l_1}\rrbracket_{bpe\,\mathcal{M}} & \cup\ \mathcal{G}\llbracket t_2^{l_2}\rrbracket_{bpe\,\mathcal{M}} \\
 & & & \cup\ \{\mathsf{C}_{l_1,bpe} \subseteq \mathsf{r}_{x,l_2}\} \\
 & & & \cup\ \{\mathsf{C}_{l_2,bpe[x\mapsto l_2]} \subseteq \mathsf{C}_{l,bpe}\} \\
\mathcal{G}\llbracket \texttt{(if } t_0^{l_0}\ t_1^{l_1}\ t_2^{l_2}\texttt{)}^{l}\rrbracket_{bpe\,\mathcal{M}} & = & \mathcal{G}\llbracket t_0^{l_0}\rrbracket_{bpe\,\mathcal{M}} & \cup\ \mathcal{G}\llbracket t_1^{l_1}\rrbracket_{bpe\,\mathcal{M}}\} \\
 & & & \cup\ \{\mathsf{C}_{l_1,bpe} \subseteq \mathsf{C}_{l,bpe}\} \\
 & & & \cup\ \mathcal{G}\llbracket t_2^{l_2}\rrbracket_{bpe\,\mathcal{M}}\} \\
 & & & \cup\ \{\mathsf{C}_{l_2,bpe} \subseteq \mathsf{C}_{l,bpe}\} \\
\mathcal{G}\llbracket \texttt{(cons } t_1^{l_1}\ t_2^{l_2}\texttt{)}^{l}\rrbracket_{bpe\,\mathcal{M}} & = & \mathcal{G}\llbracket t_1^{l_1}\rrbracket_{bpe\,\mathcal{M}} & \cup\ \mathcal{G}\llbracket t_2^{l_2}\rrbracket_{bpe\,\mathcal{M}} \\
 & & & \cup\ \{\{(l_1, l_2, bpe)\} \subseteq \mathsf{C}_{l,bpe}\} \\
\mathcal{G}\llbracket \texttt{(car } t_0^{l_0}\texttt{)}^{l}\rrbracket_{bpe\,\mathcal{M}} & = & \mathcal{G}\llbracket t_0^{l_0}\rrbracket_{bpe\,\mathcal{M}} & \cup\ \{\{(l_1, l_2, bpe_0)\} \subseteq \mathsf{C}_{l_0,bpe} \implies \mathsf{C}_{l_1,bpe_0} \subseteq \mathsf{C}_{l,bpe} \\
 & & & \qquad \mid (l_1, l_2, bpe_0) \in \mathsf{Cons}_*\} \\
\mathcal{G}\llbracket \texttt{(cdr } t_0^{l_0}\texttt{)}^{l}\rrbracket_{bpe\,\mathcal{M}} & = & \mathcal{G}\llbracket t_0^{l_0}\rrbracket_{bpe\,\mathcal{M}} & \cup\ \{\{(l_1, l_2, bpe_0)\} \subseteq \mathsf{C}_{l_0,bpe} \implies \mathsf{C}_{l_2,bpe_0} \subseteq \mathsf{C}_{l,bpe} \\
 & & & \qquad \mid (l_1, l_2, bpe_0) \in \mathsf{Cons}_*\} \\
\mathcal{G}\llbracket \texttt{(defvar } x\ t_0^{l_0}\ p\texttt{)}\rrbracket_{bpe\,\mathcal{M}} & = & \mathcal{G}\llbracket t_0^{l_0}\rrbracket_{bpe\,\mathcal{M}} & \cup\ \{\mathsf{C}_{l_0,bpe} \subseteq \mathsf{r}_{x,bpe(x)}\} \\
 & & & \cup\ \mathcal{G}\llbracket p\rrbracket_{bpe\,\mathcal{M}} \\
\mathcal{G}\llbracket \texttt{(defun } x_0\ \texttt{(}x_1\texttt{) } e\texttt{) } p\rrbracket_{bpe\,\mathcal{M}} & = & \{\{\texttt{(func (}x_1\texttt{) } e\texttt{)}\} \subseteq \mathsf{r}_{x_0,\diamond}\} & \cup\ \mathcal{G}\llbracket p\rrbracket_{bpe\,\mathcal{M}}
\end{array}
$$

Figure 3: Generating Constraints.

The constraint generation rules in Figure 3 are mostly straightforward translations of the corresponding rules of the acceptability relation.

The most involved case is the treatment of procedure applications. In addition to the generation of constraints for the terms at the operator position and the parameter position, every procedure flowing into the operator triggers the generation of constraints for its body under the current birthplace environment via a conditional constraint.

The treatment of the primitives `car` and `cdr` works in a similar way: we do not know, which abstract pairs could occur at the operator position. Therefore, the anaylsis generates conditional constraints for all abstract pairs.

The well-definedness and the termination of the algorithm follow by simple fix-point arguments on a underlying finite complete lattice.

$\mathcal{G}\llbracket p\rrbracket_{bpe\,\mathcal{M}}$ as specified generates a large number of conditional constraints in the application and `car` and `cdr` rules, many of which are never triggered during the constraint-solving phase. Therefore, our implementation defers the generation of their right-hand sides until constraint solving. It would have been possible to specify the analysis this way from the beginning, but this would mean having to mix the constraint-generation and constraint-solving phase, which would obscure the presentation.

7.2 Solving the Constraints

The generated set of constraints express the behavior of all valid dynamic scope analyses. To get the least dynamic scope analysis, we close the generated constraints under the following inference rules $\mathcal{S}$:

$$
[\textbf{VarProp}] \quad \frac{\{a\} \subseteq \mathsf{V}_0 \quad \mathsf{V}_0 \subseteq \mathsf{V}_1}{\{a\} \subseteq \mathsf{V}_1}
$$

$$
[\textbf{CondProp}] \quad \frac{\{a\} \subseteq \mathsf{V} \quad \{a\} \subseteq \mathsf{V} \implies \mathsf{co}}{\mathsf{co}}
$$

and write $\mathcal{S}(\mathsf{CO})$ for the closure of a set of constraints CO under $\mathcal{S}$.

The actual dynamic scope analysis results from the solving phase as all abstract values associated with a variable V after generating the initial constraints and closing those constraints under $\mathcal{S}$:

$$
dsa(p)(\mathsf{V}) = \{a \mid \{a\} \subseteq \mathsf{V} \in \mathcal{S}(\mathcal{G}\llbracket p\rrbracket_{bpe_\diamond\,\emptyset})\}
$$

where $bpe_\diamond$ denotes the top-level birthplace environment.

For our implementation, we employ the standard technique of using a graph representation for the constraint set and apply a worklist algorithm on the graph to compute the least solution of the original constraints [2, 14, 32].

Package	Lines	Prims	Bps	Dynamic Bps	Iters	Analysis Time (sec)
`mail-utils.el`	355	51	63	0	4159	0.96
`rfc822.el`	378	48	56	1	89428	81.84
`add-log.el`	718	74	67	1	22284	8.32
`pop3.el`	839	67	169	5	93640	130.49
`footnote.el`	975	47	153	0	115930	73.86

Figure 4: Analyzed Emacs Lisp packages, their size in lines of code after macro expansion, the number of additionally used primitives, the number of birthplaces, the number of birthplaces recognized as dynamic binding, the number of iterations the worklist algorithm used, and the analysis time.

The worklist algorithm always terminates. Every program induces only a finite set of abstract values ($\mathbf{Abs}_*$) and there is only a finite number of potential nodes since there is only a finite number of program points, variables, and birthplace environments. Hence, the analysis propagates a finite number of data objects over a finite number of nodes. The process ends after a finite number of steps: at the latest when every datum has arrived at every node.

The algorithm has exponential worst-case complexity with respect to the size of the analyzed program: the number of all possible birthplace environments is already exponential. However, the next section shows that our prototype implementation is already practical for medium-sized real-world examples. Also, since the translation of Emacs Lisp programs into a new substrate ideally happens only once, speed is not quite as important as, say, in compilers which run often. Instead, precision is at a premium.

8. MEETING THE REAL WORLD

Our prototype implementation of the algorithm is in about 5500 lines of Scheme code and runs atop Scheme 48 [15], a byte-code implementation of Scheme. It handles a large subset of Emacs Lisp programs. Specifically, it correctly deals with a number of aspects of the language not treated in this paper including the following:

- multi-parameter functions and optional arguments,
- `catch` and `throw`,
- `funcall` and the duality between functions and their names, and
- separation of function and value components of bindings.

In this section, we present the results of the analysis run on various packages taken directly from the XEmacs package collection. To receive accurate information from real packages, the implementation must know the flow behavior of a substantial number of used XEmacs's primitives.

The implementation contains a small macro language to describe the flow behavior of basic primitives for which no implementation in Emacs Lisp exists. Using those macros simplifies the description of the primitives tremendously. For instance, the three lines

```
(primitive-flow (FILENAME)
                ((const) (union (symbol t)
                                (symbol nil))))
```

describes the constraint generation for the built-in primitive `file-exists-p` that checks for the existence of a file with a given name.

Currently, the system emits an annotated version of the input program, marking those bindings which would have to stay dynamic under lexical binding. The *binding-type condition* which we use to decide to which type a variable reference belongs, is the following:

Binding-type condition *A variable is used dynamically iff the abstract cache registers some abstract object for the variable under its label for a* different *birthplace than the static one, that is iff*

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

Further conditions exist for our implementation to recognize Emacs Lisp `let`s used in the flavor of statically scoped `let*`'s or `letrec`'s in Scheme.

The results in this section were obtained by running the implementation under the Scheme system Scheme 48 0.53 on an Athlon 1 GHz system with 256 kByte second-level cache and 256 MByte of physical memory. We did not put any effort to highly optimize or to compile our implementation to native code; feasibility was our main concern.

Figure 4 lists the packages used for the experimental results. They are all part of the regular XEmacs distribution. `Mail-utils` contains utility functions used both by the other packages `rmail` and `rnews`. `Rfc822` implements a parser for the standard internet messages. `Pop3` provides POP3 functionality for email clients. `Add-log` lets a programmer manage files of changes for programs. `Footnote` offers the functionality to add footnotes to XEmacs buffers.

Figure 4 shows that the analysis is highly accurate: it only leaves behind a small number of dynamic binding constructs.

9. LIMITATIONS

While the analysis described here solves some of the hardest problems associated with translating Emacs Lisp programs to readable Scheme programs, a few remain:

`eval` Emacs Lisp has an `eval` function which interprets a piece of data as an Emacs Lisp expression. Its semantics is naturally quite undefined in a Scheme environment. Except for simple cases (for example, where the expression to be evaluated is a symbol), there is no idiomatic translation for `eval` forms. Programmers must transform the Emacs Lisp code not to use `eval` before attempting translation. Dynamic generation of symbols as well as some introspection capabilities of the language also belong in this category.

buffer-local variables Emacs Lisp features *buffer-local variables* which implicitly change value according to the current buffer. This an unfortunate conflation of the language semantics and the application domain, and often yields to unexpected and hard-to-track behavior of Emacs Lisp code. However, buffer locality is usually a *global* property of variables—programs rarely use the same variable both buffer-locally and buffer-globally. Hence, a feasible approach is to translate buffer-local variables into special designated data structures and access them via special constructs rather than preserving their implicit nature. No special analysis is required as long as the calls to `make-variable-buffer-local` are close to their variable declarations.

Note that these kinds of problems are inherent in almost *any* translation from one programming language to another, if maintainability is to be preserved.

10. RELATED WORK

10.1 Dynamic Binding

Despite the fact that languages with dynamic variable binding have existed for a long time, formulations of semantics for these languages are quite rare. On the other hand, it is folklore that dynamic binding can be eliminated by a *dynamic-environment passing translation* that makes the dynamic bindings explicit [24].

Gordon [9] initially formalized dynamic binding in the context of early Lisp dialects and studied their metacircular interpreters, using denotational semantics.

Moreau [20] rounded up the view on dynamic binding by introducing a syntactic theory of dynamic binding with a calculus allowing equational reasoning. From this theory, he also derived a small-step semantics using evaluation contexts and syntactic rewriting as developed by Felleisen and Friedman [6]. Wright and Felleisen [30, 31] and Harper and Stone [11] formulated semantics for exception mechanisms which also employ a kind of dynamic binding.

Lewis at al. [17] introduce a language feature called *implicit parameters* that provides dynamically scoped variables for languages with Hindley-Milner type systems, and formalize it with an axiomatic semantics. However, functions with implicit parameters are not first-class values in their setting.

10.2 Flow Analysis

Most realistic implementations of flow analysis for functional programming languages are simple monovariant (or 0-CFA) flow analyses [12, 7, 8, 26], that is, the analysis looks at every program point independent of context.

Shivers [27] proposed the splitting of the analysis at a function call sites depending on the context of the last recent k procedure calls (called k-CFA). Other splittings, also depending on procedure calls, were proposed by Jagannathan and Weeks [14] as poly-k-CFA and by Wright and Jagannathan [32] as *polymorphic splitting*.

The concept of coinduction arose from Milner and Tofte's works [18] on semantics and type systems of an extended λ-calculus with references. Nielson and Nielson were the first to use coinduction as a means for specifying a static analysis [22]. Their work provides the theoretical framework for the specification of our analysis.

10.3 Subject reduction

The notion of correctness we used is generally called a *subject reduction* result. Curry and Feys [5] introduced subject reduction to show the correctness of predicates in the languages of combinatory logic. Mitchell and Plotkin [19] used the idea to show a type correctness result for a λ-calculus like language, whereas Wright and Felleisen [30, 31] adapted it to the more flexible concept of operational semantics with reduction rules and evaluation contexts. Wright and Jagannathan [32] used the same technique for their *polymorphic splitting* flow analysis.

10.4 Emacs Lisp and Scheme

A number of other projects have built or are currently building Scheme-based variants of Emacs. The oldest is Matt Birkholz's Emacs Lisp interpreter which allows running Emacs Lisp programs on top of MIT Scheme's Edwin editor. Current efforts include Ken Raeburn's work on creating a Guile-based Emacs [25], the Guile Emacs project [10] as well as Per Bothner's JEmacs [3, 4] which aims at re-implementing Emacs atop Java bytecodes, leveraging Bothner's Kawa compiler for Scheme. As far as Emacs Lisp is concerned, only JEmacs seems to have seen significant work as far as making Emacs Lisp programs run. None of these projects address permanently translating Emacs Lisp code to Scheme while retaining maintainability.

11. CONCLUSION AND FUTURE WORK

We have specified, proved correct and implemented a flow analysis for Emacs Lisp whose distinguishing feature is its correct handling of dynamic binding. The primary purpose of the analysis is to aid translation of Emacs Lisp programs into more modern language substrates with lexical scoping since most binding in real Emacs Lisp programs behaves identically under lexical and dynamic scoping. Our analysis is highly accurate in practice. Our prototype implementation is reasonably efficient.

We have two main directions for future research:

- Improving the efficiency of the analysis by ordinary optimization, compilation code and modularization of the constraints [8], and
- integration of the analysis into a translation suite from Emacs Lisp to Scheme.

Acknowledgments. We would like to thank the initial members of the `el2scm` project: Martin Gasbichler, Johannes Hirche, and Peter Biber. Specifically, Peter Biber developed a precursor to the analysis presented here, demonstrating the feasibility of the project. Peter Thiemann provided valuable suggestions for the paper. We also thank the ICFP referees for valuable comments.

12. REFERENCES

[1] A. Aiken. Set constraints: Results, applications and future directions. *Lecture Notes in Computer Science*, 874:326–335, 1994.

[2] A. Aiken and E. Wimmers. Type inclusion constraints and type inference. In *Proceedings of the FPCA 1993*, pages 31–41, 1993.

[3] P. Bothner. JEmacs-the java/scheme-based emacs. In *Proceedings of the FREENIX Track: 2000 USENIX Annual Technical Conference (FREENIX-00)*, pages 271–278, Berkeley, CA, June 18–23 2000. USENIX Ass.

[4] P. Bothner. JEmacs—the Java/Scheme-based Emacs text editor. `http://jemacs.sourceforge.net/`, Feb. 2001.

[5] H. B. Curry and R. Feys. *Combinatory Logic*, volume I. North-Holland, Amsterdam, 1958.

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

[7] C. Flanagan and M. Felleisen. Set-based analysis for full scheme and its use in soft-typing. Technical Report TR95-254, Rice University, Oct., 1995.

[8] C. Flanagan and M. Felleisen. Componential set-based analysis. *ACM Transactions on Programming Languages and Systems*, 21(2):370–416, Mar. 1999.

[9] M. J. C. Gordon. *Programming Language Theory and its Implementation*. Prentice-Hall, 1988.

[10] Guile Emacs. `http://gemacs.sourceforge.net/`, July 2000.

[11] R. Harper and C. Stone. An interpretation of Standard ML in type theory. Technical Report CMU-CS-97-147, Carnegie Mellon University, Pittsburgh, PA, June 1997. (Also published as Fox Memorandum CMU-CS-FOX-97-01.).

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

[13] R. Hindley and J. Seldin. *Introduction to Combinators and λ-Calculus*, volume 1 of *London Mathematical Society Student Texts*. Cambridge University Press, 1986.

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

[15] R. A. Kelsey and J. A. Rees. A tractable Scheme implementation. *Lisp and Symbolic Computation*, 7(4):315–335, 1995.

[16] B. Lewis, D. LaLiberte, R. Stallman, and the GNU Manual Group. GNU Emacs Lisp reference manual. `http://www.gnu.org/manual/elisp-manual-20-2.5/elisp.html`, 1785.

[17] J. Lewis, M. Shields, E. Meijer, and J. Launchbury. Implicit parameters: Dynamic scoping with static types. In *Proceedings of the 27th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Boston, Massachusetts*, pages 108–118, Jan 2000.

[18] R. Milner and M. Tofte. Co-induction in relational semantics. *Theoretical Computer Science*, 87:209–220, 1991.

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

[20] L. Moreau. A Syntactic Theory of Dynamic Binding. *Higher-Order and Symbolic Computation*, 11(3):233–279, Dec. 1998.

[21] M. Neubauer. Dynamic scope analysis for Emacs Lisp. Master's thesis, Eberhard-Karls-Universität Tübingen, Dec. 2000. `http://www.informatik.uni-freiburg.de/~neubauer/diplom.ps.gz`.

[22] F. Nielson and H. R. Nielson. Infinitary control flow analysis: a collecting semantics for closure analysis. In *Proc. POPL'97*, pages 332–345. ACM Press, 1997.

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

[24] C. Queinnec. *Lisp in Small Pieces*. Cambridge University Press, 1996.

[25] K. Raeburn. Guile-based Emacs. `http://www.mit.edu/~raeburn/guilemacs/`, July 1999.

[26] M. Serrano and M. Feeley. Storage use analysis and its applications. In *Proceedings of the 1fst International Conference on Functional Programming*, page 12, Philadelphia, June 1996.

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

[28] R. Stallman. GNU extension language plans. Usenet article, Oct. 1994.

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

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

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

[32] A. K. Wright and S. Jagannathan. Polymorphic splitting: an effective polyvariant flow analysis. *ACM Transactions on Programming Languages and Systems*, 20(1):166–207, Jan. 1998.