Flow-Sensitive Control-Flow Analysis in Linear-Log Time

Michael D. Adams

Submitted to the faculty of the University Graduate School
in partial fulfillment of the requirements
for the degree
Doctor of Philosophy
in the School of Informatics and Computing,
Indiana University
October 2011

Accepted by the Graduate Faculty, Indiana University, in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

Doctoral Committee

R. Kent Dybvig, Ph.D.

(Principal Advisor)

Amr Sabry, Ph.D.

Amal Ahmed, Ph.D.

Daniel Leivant, Ph.D.

September 30, 2011

To my mother who instilled in me an insatiable desire for learning,
and to my father who taught me the fundamentals of computing
before I could even read.

Acknowledgements

Many thanks go to my parents, David L. Adams and Mary C. Adams, for teaching me to learn and fostering my interests at a young age.

Many thanks go to my advisor, R. Kent Dybvig, for guiding me through the doctoral process and teaching me the ways of academia. A special thanks goes to him for asking me to implement the optimization that started this line of research.

I'd also like to thank the other members of my doctoral committee: Amr Sabry, Amal Ahmed, and Daniel Leivant. Each of them has been instrumental in my graduate education.

This dissertation is an extended version of Adams et al. [2011] and due thanks go to my co-authors, Andy W. Keep, Jan Midtgaard, Matt Might, Arun Chauhan, and R. Kent Dybvig, for the efforts they put into preparing that paper. Comments from Dan Friedman, Jeremiah Willcock, Lindsey Kuper, Alex Hearn, and anonymous reviewers led to several improvements in that paper's presentation.

Many thanks go to Andy W. Keep who was a co-developer of the system described in this dissertation. Of particular note is his work on benchmarking the system described in this dissertation. Thanks also go to Jan Midtgaard for his work in developing and proving the semantics of the system described in this dissertation. Thanks go to Matt Might for introducing me to the fundamentals of control-flow analysis.

Thanks go to David S. Wise who was mentor to me in my first years as a graduate student. His passion always made research enjoyable.

Acknowledgements

Thanks go to Daniel P. Friedman for introducing me to the field of programming languages and for his advice about graduate school, academia and writing.

Thanks go to Amr Sabry for the time he spent as my advisor and his willingness to discuss my research ideas.

Thanks go to the PL Wonks for their support and forbearance with my early presentations of this material.

Thanks go to Roshan P. James and William E. Byrd for being friends and helping me survive graduate school.

Preface

History

The work that led to the research in this dissertation initially started as a summer project. Dr. Dybvig, who is now my advisor, asked Andy Keep and me if we were interested in developing an optimization that eliminates unnecessary dynamic type checks. This was an optimization that he had wanted to add to Chez Scheme for some time. By default, Chez Scheme includes type checks for most primitive operations even if it is impossible for a particular type check to fail.[1] These checks can be disabled but at the cost of run-time type safety. The goal of the optimization is to eliminate those type checks that are redundant or otherwise unnecessary but keep the type checks that are necessary to ensure type safety. An example of this is the following expression.

```
(let ([x (read)])
  (display (car x))
  (display (cdr x)))
```

The type check in `cdr` will always succeed, because if `x` is not a pair, the type check in `car` fails before `cdr` is called. Another example is the following.

```
(let ([x (read)])
  (if (pair? x)
    (car x)
    (raise-error)))
```

[1] Chez Scheme will eliminate type checks when arguments to a primitive are constants or a function application calls a known function. These cases may arise through other optimizations such as constant propagation and inlining.

Due to the explicit `pair?` check, the implicit type check in `car` always succeeds. The idea of the optimization is to eliminate the unnecessary type checks in each of these expressions and thus produce more efficient code.

Dr. Dybvig had one more requirement the optimization had to satisfy. The optimization had to run in better than $O(n^2)$ time, where n is the size of the program being analyzed. The reason for this requirement is a matter of design philosophy. An optimization ought to not only produce efficient code but also process programs quickly. In addition, metaprogramming features such as macros, compiler compilers, or interface generators may result in input programs that are large or unusually structured. These inputs can have sizes and structures that no programmer would normally write. In order to process these programs quickly, our optimization must not only be fast on large common-case inputs but also fast on large worst-case inputs. In these cases, being fast in practice is no substitute for being fast always.

While this requirement may result in a compiler having to forgo powerful optimizations if they are costly to run, it ensures that every optimization runs quickly. Thus, the compiler can afford to run more optimizations and still take less time than if it used more expensive optimizations. Any one optimization might have difficulty handling a particular case, but in the group of optimizations at least a few of them are likely to succeed. By having "lots of little bullets" rather than one silver bullet, the overall result of optimization is likely to be better [Peyton Jones et al., 1996].

This constraint on the running time of the optimization was an essential driving force for the research. Optimization and static-analysis research often focuses on improving the precision of the analysis or optimization at the cost of running time. When running time *is* considered, it is often only in terms of the common case. Worst-case scenarios are dismissed as "code that no programmer would ever write." Requiring the asymptotic bound on compilation time gave a fresh perspective. Rather than aiming for precision in

the worst case and efficiency in the common case, we aimed for efficiency in the worst case and precision in the common case.

Lessons

A number of lessons and techniques for research arose through this process, and I would like to share these in hopes they might help other students pursuing a research career.

Look where others are not. The vast majority of research on program analysis focuses on improving the precision of the analysis. While most research gives at least a passing nod to common-case efficiency, worst-case asymptotic performance is largely ignored. By making asymptotic performance a primary goal, I found results that others had not.

Take a solved problem and add an asymptotic bound. Though problems like control-flow analysis (CFA) are well studied in terms of precision, their asymptotic behavior is often analyzed only after the fact. That is to say, research often focuses first on improving the precision of the analysis, and only afterwards attempts to improve the asymptotic performance. The natural result is an analysis with poor asymptotic performance. An alternative is to focus primarily on asymptotic performance and only increase the precision when doing so does not break a predetermined asymptotic bound. The result is likely to be novel, since most research attacks the problem from the other direction. The result is also likely to be non-trivial as a fair amount of algorithmic work is likely to be needed in order to obtain the desired asymptotic bound. Examples of these sorts of results include the work on sub-0CFA [Ashley and Dybvig, 1998], CP0 [Waddell and Dybvig, 1997], and the research in this dissertation.

Take a solved problem and scale it up to a real-world implementation. Many elegant solutions exist for idealized languages, but real-world problems often have interesting constraints not present in an idealization. For example, Scheme has `case-lambda`, mutable variables, and other language features not usually present in idealized systems. If we do research in an abstract, idealized setting, then it is tempting to omit such forms if they lead to complications. By working on a real-world implementation, we are forced to deal with these hard cases, and these hard cases can lead to novel and creative solutions.

Look for real-world problems with interesting constraints. In the case of this research, solving the problem in better than $O(n^2)$ time served as the constraint that lead to the most interesting results. In the creative arts it is widely recognized that boundaries and constraints can help stimulate creativity and make creativity easier than it would be in their absence. Likewise, real-word constraints and the hard cases that come with them can help guide research. They provide both a metric by which the research may be judged and an assurance that the research is relevant to the worlds needs. At this point in computer science, it is hard to find topics that are not already well researched in an idealized setting, but many real-world problems have yet to be solved.

Have a fallback result. Hard problems that are not already solved may be impossible to solve. Thus, when starting a line of research, it is helpful to have fallback solutions that may not be perfect but do partially solve the problem. For example, early in this research many imperfect solutions were considered such as bounding the span over which certain types of information are collected. These solutions lead to analyses that were not as precise as desired, but were both sound and novel. While in the end these solutions were not used, having them available provided the freedom to dare approach the harder, more precise version of the analysis. If the harder analysis did not work out, the fallback solutions were always available.

Have desiderata that can be sacrificed. One has to be careful about the constraints imposed on a problem. It is easy to over constrain a problem and make it impossible to solve. To avoid this, treat some constraints as desiderata rather than requirements. For example, in the present research, soundness and performance were hard requirements. However, though precision of the analysis was desired, we were willing to sacrifice precision in order to gain performance. This was especially useful for features like mutable variables where we had to give up on collecting restrictive information (see Section 6.7) in order to maintain our desired time bound. This gave up some precision, but allowed the analysis to achieve the desired performance.

Michael D. Adams

Flow-Sensitive Control-Flow Analysis in Linear-Log Time

The flexibility of dynamically typed languages such as JavaScript, Python, Ruby, and Scheme comes at the cost of run-time type checks. Some of these checks can be eliminated via control-flow analysis. However, traditional control-flow analysis (CFA) is not ideal for this task as it ignores flow-sensitive information that can be gained from dynamic type predicates, such as JavaScript's `instanceof` and Scheme's `pair?`, and from type-restricted operators, such as Scheme's `car`. Yet, adding flow-sensitivity to a traditional CFA worsens the already significant compile-time cost of traditional CFA. This makes it unsuitable for use in just-in-time compilers.

In response, this dissertation presents a fast, flow-sensitive type-recovery algorithm based on the linear-time, flow-insensitive sub-0CFA. The algorithm has been implemented as an experimental optimization into Chez Scheme compiler, where it has proven to be effective, justifying the elimination of about 60% of run-time type checks in a large set of benchmarks. The algorithm processes on average over 100,000 lines of code per second and scales well asymptotically, running in only $O(n \log n)$ time. This compile-time performance and scalability is achieved through a novel combination of data structures and algorithms.

R. Kent Dybvig, Ph.D.
(Principal Advisor)

Amr Sabry, Ph.D.

Amal Ahmed, Ph.D.

Daniel Leivant, Ph.D.

Contents

1. Introduction **1**

2. CFA **5**

2.1. CFA 5

2.1.1. 0CFA for the λ-calculus 5

2.1.2. 0CFA for the extended λ-calculus 8

2.2. Flow-graph implementation of CFA 12

2.3. Top and escaped functions 14

2.4. Sub-0CFA 15

2.5. Non-function types 18

3. Flow Sensitivity **19**

3.1. Flow-sensitivity for unconditional observers 21

3.2. Flow-sensitivity for conditional observers 24

3.3. Flow-graph representation of flow-sensitivity 29

4. Fast Flow Sensitivity **31**

4.1. Context skipping 33

4.2. Context skipping and reachability 42

4.3. Selecting context skips 44

4.4. Caching context skips 47
4.5. Algorithm Summary 54

5. Example **58**

5.1. Code 58
5.2. Abstract syntax tree 58
5.3. Value graphs 61
5.4. Context cache 64
5.5. Variables 66
5.5.1. Flow graph for `x` 66
5.5.2. Flow graph for `y` 70
5.5.3. Flow graph for `z` 74
5.5.4. Other variables 77
5.6. Result 77

6. Implementation **79**

6.1. Implementation structure 79
6.2. Implementation notes 80
6.3. Order of evaluation 80
6.4. Local binding 82
6.5. Lambda 83
6.6. Smart primitives 84
6.7. Mutable variables 85

7. Benchmarks **87**

7.1. Effectiveness 87
7.2. Efficiency 89

8. Related Work **94**

8.1. CFA and CFA-based type recovery 94

8.2. Type recovery based on type inference 95

8.3. Recent type-recovery applications 96

8.4. Other related work 97

9. Future Work **99**

9.1. User-defined types 99

9.2. Objects and rich types 100

9.3. Asymptotic improvements 101

9.4. Loops and general control flow 102

9.5. Static single assignment form 102

9.6. Sub-kCFA 103

9.7. Generalized skipping functions 103

9.8. Backwards abstract interpretation 105

10. Conclusion **107**

A. Source Code Overview **109**

A.1. Using the code 109

A.2. Algorithm 110

A.2.1. Preprocessing 110

A.2.2. Flow-graph construction 111

A.2.2.1. Environmental flow-graph nodes 111

A.2.2.2. Value flow-graph nodes 112

A.2.3. Post-processing 113

A.3. Data types . . . 113
A.3.1. Trex records . . . 113
A.3.2. Abstract syntax trees . . . 114
A.3.3. Flow graphs . . . 115
A.3.4. AST flow graphs . . . 117
A.3.5. Identifiers . . . 117
A.3.6. Types . . . 117
A.3.7. Environmental flow functions . . . 118
A.4. Auxiliary code . . . 118
A.4.1. datatype.ss . . . 118
A.4.2. records-io.ss . . . 120
A.4.3. define-ignored.ss . . . 120
A.4.4. iota.ss . . . 121
A.4.5. safe-forms.ss . . . 121
A.4.6. syntax-helpers.ss . . . 121
A.5. Third-party code . . . 121
A.5.1. srfi-39.ss . . . 121
A.5.2. match.ss . . . 122

B. Source Code **123**

B.1. Software license . . . 123
B.2. Algorithm . . . 124
B.2.1. Preprocessing . . . 133
B.2.2. Graph construction . . . 144
B.2.3. Post-processing . . . 162

B.3. Data types 166
B.3.1. Abstract syntax trees 166
B.3.2. Flow graphs 171
B.3.3. Types 179
B.4. Auxiliary code 197
B.5. Third-party Code 204
B.5.1. SRFI 39 204
B.5.2. Match 205

Bibliography **222**

Indexes **231**

Index of Source Files 231
Index of Functions 232
Index of Macros 239
Index of Record Types 241

Vita **242**

List of Figures

2.1. Concrete semantics of λ-calculus 6

2.2. 0CFA constraint rules 7

2.3. Concrete semantics for the extended λ-calculus 9

2.4. Constraint rules for extended λ-calculus 11

2.5. Flow graphs in 0CFA 13

2.6. Flattened lattice of functions 17

3.1. Common definitions 21

3.2. Definitions for unconditional observers 24

3.3. Rules for unconditional observers 25

3.4. Predicate aware definitions 26

3.5. Predicate aware rules 27

3.6. Flow graphs for flow-sensitive 0CFA 29

4.1. True and false expression guards 33

4.2. Context skipping function 37

4.3. Graph form of canonical skipping functions 42

4.4. Example AST for skipping context selection 45

4.5. Example of quadratic $\mathcal{V}_{C,e}$ calculation 47

4.6. Layered structure of the $\mathcal{V}_{C,e}$ cache 49

4.7. A perfectly balanced skip list 51
4.8. Skip lists sharing a tail 52
4.9. A Myers stack 52
4.10. Example AST 56
4.11. Example AST with skips 56
4.12. Worst case for Myers stacks 57

5.1. Example expression 59
5.2. Abstract syntax tree 60
5.3. Value flow 62
5.4. Cache of contexts 65
5.5. Context skipping for `x` 67
5.6. Flow graph for `x` 69
5.7. Context skipping for `y` 71
5.8. Flow graph for `y` 72
5.9. Context skipping for `z` 75
5.10. Flow graph for `z` 76

6.1. Rules for `let` and `letrec` 81

7.1. Average percent of type checks removed 88
7.2. Percent of type checks removed 91
7.3. Percent of type checks removed 92
7.4. Source node count versus analysis time 93

9.1. Lattice for rich types 100

1. Introduction

Dynamically typed languages such as JavaScript, Python, Ruby, and Scheme are flexible, but this flexibility comes at the cost of type checks at run time. This cost can be reduced via type-recovery analysis [Shivers, 1991], which attempts to discover variable and expression types at compile time and thereby justify the elimination of run-time type checks.

Since these are higher-order languages in which the call graph is not static, the type-recovery analysis generally must be a form of control-flow analysis [Shivers, 1988]. A control-flow analysis (CFA) tracks the flow of function values to call sites and builds the call graph even as it tracks the flow of other values to their use sites.

To maximize the number of checks removed, the analysis must take evaluation order into account. That is, it must be *flow sensitive*[1] [Banning, 1979]. In the following expression, even a flow-insensitive control-flow analysis can determine that `x` is a pair and thus `car` need not check that `x` is a pair.

```
(let ([x (cons e1 e2)]) (car x))                    (1.1)
```

To make a similar determination in the following expressions, however, evaluation order must be taken into account as the `read` function can return any type of value.

```
(let ([x (read)]) (begin (cdr x) (car x)))          (1.2)
(let ([x (read)]) (if (pair? x) (car x) #f))        (1.3)
```

[1] This use of the term *flow sensitive* agrees with the original definition of Banning [1979], in which an analysis takes order of evaluation into account, as well as with a glossary definition of Mogensen [2000], in which separate results are computed for distinct program points. Our analysis might also be considered path sensitive, depending on the definition of path sensitivity used.

Because it does not take order of evaluation into account, a flow-insensitive analysis treats all references the same. On the other hand, a flow-sensitive analysis can determine that `(car x)` is reached in Example 1.2 only after passing the implicit pair check in `(cdr x)` and in Example 1.3 only when the explicit `pair?` check succeeds. Thus, the implicit pair check in `car` can be eliminated in both expressions.

In general, the analysis cannot eliminate all type checks. It can eliminate only those that are redundant or otherwise unnecessary to preserve the semantics of the program. In Example 1.2, the type check in `cdr` must be preserved in order to preserve the error semantics of the expression, but the type check in `car` does not affect the semantics of the expression and is thus redundant.

A type-recovery analysis is used to optimize code and not to statically verify the type soundness of code. Thus, the analysis is not allowed to reject a program even if the program might be considered not well typed. Also, the analysis must handle the complete language "as is" and cannot require type annotations or other assistance from the programmer.

On the other hand, being used solely for optimization simplifies matters in other ways. Since the analysis never rejects a program, no error messages need be displayed to the user. Likewise, an optimization has the option of using conservative approximations and heuristics that would not be appropriate in a type checker. This freedom to conservatively approximate gives us the freedom to adjust the analysis in ways not available to a type checker and thus reduce the cost of the analysis or increase its effectiveness.

This dissertation presents a flow-sensitive control-flow analysis that runs in linear-log time and is suitable for use in a type-recovery optimization. Because the analysis is intended to justify type recovery in a fast production compiler [Dybvig, 2010], it is based on sub-0CFA [Ashley and Dybvig, 1998], a linear-time, flow-insensitive variant of 0CFA [Shivers, 1988]. The analysis uses a novel combination of data structures and algorithms to add flow sensitivity to sub-0CFA at the cost of only an additional logarithmic factor.

The analysis has been implemented as an experimental optimization for the commercial Chez Scheme compiler, where it has proven to be effective and justifies the elimination of about 60% of run-time type checks. The algorithm has also proven to be fast, processing over 100,000 lines of code per second on average. Furthermore, since it runs in $O(n \log n)$ time, it scales well to large input programs.

Since this analysis is to be used by an optimization in an industrial compiler, the perspective taken with this research is to favor performance over precision. By having "lots of little bullets" rather than one silver bullet, the overall result of optimization is likely to be better [Peyton Jones et al., 1996]. Rather than aiming for precision in the worst case and efficiency in the common case, the analysis aims for efficiency in the worst case and precision in the common case.

Of course, the optimization must still preserve the semantics of the program being optimized. Thus, the analysis must *conservatively* approximate the behavior of the program being optimized. For example, both the order of evaluation and the behavior of errors must be preserved. Nevertheless, as long as the analysis computes a conservative approximation, there is some flexibility in how much it approximates.

This dissertation presents two major research results. First, it presents a control-flow analysis that has a novel and strong form of flow sensitivity that elegantly handles predicates and is suitable for use in type recovery. This analysis takes $O(n^2)$ time if implemented naïvely. Second, this dissertation presents an optimized algorithm that computes the same analysis but takes only $O(n \log n)$ time.

In brief terms, the analysis handles predicates by associating two environments with the result of each expression in addition to the return value that traditional CFA associates with each expression. While associating a *single* environment with each expression is a known technique [Banning, 1979] for implementing flow sensitivity, generalizing this to two environments allows us to make the analysis predicate-aware. One environment records

the types of variables for when a predicate returns true, while the other records the types of variables for when the predicate returns false.

A naive implementation of this using standard flow-graph techniques would be quadratic in the size of the program being analyzed. This research shows how to perform this in only $O(n \log n)$ time by a combination of novel techniques and the careful engineering of known (but slightly obscure) techniques. First, this involves computing *skipping functions* to allow values to bypass the usual threading of environments through the program and thus more efficiently flow values to the locations where they are needed. Second, these skipping functions are stored in a specialized cache that allows efficient computation and re-computation of these skipping functions. Finally, the analysis carefully chooses where to use skipping functions versus the standard environment-flow rules in order to achieve the $O(n \log n)$ time bound.

The remainder of this dissertation reviews the semantics and implementation of 0CFA and sub-0CFA (Chapter 2), describes the traditional technique for implementing flow sensitivity (Chapter 3), describes a novel, efficient technique for implementing flow sensitivity (Chapter 4), works through examples of this technique (Chapter 5), discusses practical considerations in a real-world implementation (Chapter 6), presents benchmark results (Chapter 7), reviews related work (Chapter 8), outlines directions for future research in this area (Chapter 9), and finally concludes (Chapter 10).

2. CFA

This chapter reviews two relevant forms of control-flow analysis, Shivers's 0CFA and Ashley's sub-0CFA. It also discusses their implementations in terms of flow graphs, how top and escaped values are handled, and the representation of non-function types. Readers familiar with control-flow analysis may wish to skip forward to Chapter 3.

2.1. CFA

2.1.1. 0CFA for the λ-calculus

Constraint rules that define 0CFA on the call-by-value λ-calculus are presented in Figure 2.2. The operational semantics of the λ-calculus is shown in Figure 2.1.

The operational semantics is mostly standard but differs from standard semantics in that, when functions are invoked, the $\texttt{app}(v_0\ v_1, \rho) :: \kappa$ context is placed on the stack instead of κ. This records the origin of the function call and makes the semantics more closely parallel the constraint rules in Figure 2.2, which implicitly track function-call origins in order to determine the output values of function calls. Though a formal proof of the soundness of the constraint rules is not included in this dissertation, such a proof is simplified by the extra context and the parallels it causes between the semantics and the constraint rules.

The analysis stores a reachability flag, $[\![e]\!]_{in}$, for each subexpression of the program being analyzed. The flag is $\top$ if the expression is reachable and $\bot$ otherwise.

$$
\begin{array}{rl}
\text{Expressions:} & e \in \mathit{Exp} = x \mid \lambda x.e \mid e\ e \\
\text{Signatures:} & s \in \mathit{State} = (\mathit{Exp} \times \mathit{Env} \times K) + (K \times \mathit{Val}) \\
& \rho \in \mathit{Env} = \mathit{Var} \rightharpoonup \mathit{Val} \\
& \kappa \in K = \mathtt{stop} \mid \mathtt{app}(\square\ e_1, \rho) :: \kappa \mid \mathtt{app}(v\ \square, \rho) :: \kappa \mid \mathtt{app}(v_0\ v_1, \rho) :: \kappa \\
& v \in \mathit{Val} = [\lambda x.e;\ \rho]
\end{array}
$$

$$
\begin{array}{rrcl}
\text{Rules (In):} & \langle x,\ \rho,\ \kappa\rangle & \rightarrow & \langle \kappa,\ v\rangle \ \text{ where } v = \rho(x) \\
& \langle \lambda x.e,\ \rho,\ \kappa\rangle & \rightarrow & \langle \kappa,\ [\lambda x.e;\ \rho]\rangle \\
& \langle e_0\ e_1,\ \rho,\ \kappa\rangle & \rightarrow & \langle e_0,\ \rho,\ \mathtt{app}(\square\ e_1, \rho) :: \kappa\rangle \\
\text{Rules (Out):} & \langle \mathtt{app}(\square\ e_1, \rho) :: \kappa,\ v_0\rangle & \rightarrow & \langle e_1,\ \rho,\ \mathtt{app}(v_0\ \square, \rho) :: \kappa\rangle \\
& \langle \mathtt{app}(v_0\ \square, \rho) :: \kappa,\ v_1\rangle & \rightarrow & \langle e,\ \rho'[x \mapsto v_1],\ \mathtt{app}(v_0\ v_1, \rho) :: \kappa\rangle \\
& & & \qquad \text{if } v_0 = [\lambda x.e;\ \rho'] \\
& \langle \mathtt{app}(v_0\ v_1, \rho) :: \kappa,\ v\rangle & \rightarrow & \langle \kappa,\ v\rangle
\end{array}
$$

Figure 2.1.: Concrete semantics of λ-calculus

In addition, for each expression, a flow variable $[\![e]\!]_{out}$ records the abstract value that flows from the expression, i.e., a subset of the lambda terms that may be returned by the expression. For example, the LAMBDA rule says that if $\lambda x.e$ is reachable, the result of that expression includes an abstract value representing the lambda.

For *Bool*, the $\sqsupseteq$ relation is the standard partial order where $\top \sqsupseteq \bot$. For $\widehat{\mathit{Val}}$, the $\sqsupseteq$ relation is the standard partial order over power sets.

All subexpressions are implicitly labeled, and all variables are uniquely alpha-renamed. Thus, the analysis distinguishes duplicate expressions and variable names.

The CALL_{in} rule ensures that the expression in the function position of an application is reachable if the application is reachable. Likewise the CALL_{mid} rule ensures that the expression in the argument position of an application is reachable if the function position of the application returns a value other than $\bot$.

Contexts: $C \in Ctxt = \Box \mid (\Box\ e_1) \mid (e_0\ \Box) \mid (\lambda x.\Box)$

Signatures:

$$\mathbb{K} \in Exp \to Ctxt$$

$$[\![e]\!]_{in} \in Bool \quad \{\text{- Whether } e \text{ is reachable -}\}$$

$$[\![e]\!]_{out} \in \widehat{Val} \quad \{\text{- What } e \text{ evaluates to -}\}$$

$$\hat{r} \in Bool = \{\bot, \top\}$$

$$\hat{v} \in \widehat{Val} = \wp(\widehat{Lam})$$

$$\widehat{Lam} = \{\lambda x_1.e_1, \lambda x_2.e_2, \lambda x_3.e_3, \dots\}$$

$$\frac{[\![\lambda x.e]\!]_{in} \sqsupseteq \top}{[\![\lambda x.e]\!]_{out} \sqsupseteq \{\lambda x.e\}}\ \textsc{lambda}$$

$$\frac{[\![e_0\ e_1]\!]_{in} \sqsupseteq \hat{r}}{[\![e_0]\!]_{in} \sqsupseteq \hat{r}}\ \textsc{call}_{in}$$

$$\frac{[\![e_0]\!]_{out} \sqsupseteq \{\hat{v}_0\} \qquad \mathbb{K}(e_0) = (\Box\ e_1)}{[\![e_1]\!]_{in} \sqsupseteq \top}\ \textsc{call}_{mid}$$

$$\frac{[\![e_0]\!]_{out} \sqsupseteq \{\lambda x.e_\lambda\} \qquad [\![e_1]\!]_{out} \sqsupseteq \{\hat{v}_1\} \qquad \mathbb{K}(e_1) = (e_0\ \Box)}{[\![e_\lambda]\!]_{in} \sqsupseteq \top \qquad [\![x]\!]_{out} \sqsupseteq \{\hat{v}_1\}}\ \textsc{call}_{fun}$$

$$\frac{[\![e_0]\!]_{out} \sqsupseteq \{\lambda x.e_\lambda\} \qquad [\![e_1]\!]_{out} \sqsupseteq \{\hat{v}_1\} \qquad [\![e_\lambda]\!]_{out} \sqsupseteq \{\hat{v}\}}{[\![e_0\ e_1]\!]_{out} \sqsupseteq \{\hat{v}\}}\ \textsc{call}_{out}$$

Figure 2.2.: 0CFA constraint rules

The CALL_{mid} and CALL_{fun} rules use $\mathbb{K}(e)$, which returns the source context[1] of e. These contexts are single-layer contexts instead of the more usual multilayer contexts, but for specifying the constraint rules only a single layer is needed. When multilayer contexts are needed, they are represented by the juxtaposition of contexts.

The CALL_{fun} rule applies when a lambda flows to a subexpression that is contextually located inside an application, i.e., $\mathbb{K}(e_1) = (e_0\ \Box)$, and a value flows to the operand of the expression, e_1. When these two conditions hold, the CALL_{fun} rule makes the body of the invoked lambda reachable and the actual argument flow to the formal parameter.

Finally, provided that the argument of a application returns, i.e., $[\![e_1]\!]_{out} \sqsupseteq \{\hat{v}_1\}$, the CALL_{out} rule takes any lambda that flows to the function position and makes its return value flow to the result of the application. This condition on e_1 prevents function calls from returning a value when either e_0 or e_1 diverge. This reflects the fact that, in those cases, the function is never called and thus cannot return.

To solve these constraints rules for a particular program, the analysis initially assigns $\bot$ to each $[\![e]\!]_{in}$ and the empty set to each $[\![e]\!]_{out}$. Then it iteratively uses the constraint rules to update $[\![e]\!]_{in}$ and $[\![e]\!]_{out}$ until they converge to a solution. In the process, $[\![e]\!]_{in}$ and $[\![e]\!]_{out}$ monotonically climb the lattices for *Bool* and $\widehat{Val}$ respectively [Nielson et al., 1999].

2.1.2. 0CFA for the extended λ-calculus

In order to bring the language in Figure 2.1 in line with the core language of Scheme, it is extend in Figure 2.3 with constants, primitives, sequencing, and conditionals. Again, the semantics is standard for a language with these features. Note that the expression $\eta(o, v_1)$ returns the result of applying a primitive function to a value.

[1] As the analysis in Figure 2.2 is extended and optimized throughout the rest of this dissertation, it will need an explicit representation of contexts to reason about. Hence for presentational purposes the contexts of expressions in the above 0CFA differ from more traditional presentations [Nielson et al., 1999].

Expressions:
$$e \in Exp = \ldots \mid c \mid e; e \mid \mathit{if}\ e\ e\ e$$
$$c \in Const = \texttt{\#f} \mid n \mid o \mid \ldots$$
$$n \in Num = 0 \mid 1 \mid 2 \mid \ldots$$
$$o \in Prim = \{\texttt{pair?}, \texttt{car}, \texttt{cdr}, \ldots\}$$

Signatures:
$$\kappa \in K = \ldots \mid \texttt{seq}(\square; e_1, \rho) :: \kappa \mid \texttt{seq}(v_0; \square, \rho) :: \kappa$$
$$\mid \texttt{cond}(\mathit{if}\ \square\ e_1\ e_2, \rho) :: \kappa \mid \texttt{cond}(\mathit{if}\ v_0\ \square\ e_2, \rho) :: \kappa$$
$$\mid \texttt{cond}(\mathit{if}\ v_0\ e_1\ \square, \rho) :: \kappa$$
$$v \in Val = \ldots \mid c$$
$$\eta \in Prim \times Val \to Val$$

Rules (In):
$$\langle c, \rho, \kappa\rangle \to \langle \kappa, c\rangle$$
$$\langle e_0; e_1, \rho, \kappa\rangle \to \langle e_0, \rho, \texttt{seq}(\square; e_1, \rho) :: \kappa\rangle$$
$$\langle \mathit{if}\ e_0\ e_1\ e_2, \rho, \kappa\rangle \to \langle e_0, \rho, \texttt{cond}(\mathit{if}\ \square\ e_1\ e_2, \rho) :: \kappa\rangle$$

Rules (Out):
$$\langle \texttt{app}(v_0\ \square, \rho) :: \kappa, v_1\rangle \to \langle \kappa, v\rangle \quad \text{if } v_0 = o \wedge v = \eta(o, v_1)$$
$$\langle \texttt{seq}(\square; e_1, \rho) :: \kappa, v_0\rangle \to \langle e_1, \rho, \texttt{seq}(v_0; \square, \rho) :: \kappa\rangle$$
$$\langle \texttt{seq}(v_0; \square, \rho) :: \kappa, v_1\rangle \to \langle \kappa, v_1\rangle$$
$$\langle \texttt{cond}(\mathit{if}\ \square\ e_1\ e_2, \rho) :: \kappa, v_0\rangle \to \begin{cases} \langle e_1, \rho, \texttt{cond}(\mathit{if}\ v_0\ \square\ e_2, \rho) :: \kappa\rangle \\ \qquad \text{if } v_0 \neq \texttt{\#f} \\ \langle e_2, \rho, \texttt{cond}(\mathit{if}\ v_0\ e_1\ \square, \rho) :: \kappa\rangle \\ \qquad \text{if } v_0 = \texttt{\#f} \end{cases}$$
$$\langle \texttt{cond}(\mathit{if}\ v_0\ \square\ e_2, \rho) :: \kappa, v_1\rangle \to \langle \kappa, v_1\rangle$$
$$\langle \texttt{cond}(\mathit{if}\ v_0\ e_1\ \square, \rho) :: \kappa, v_2\rangle \to \langle \kappa, v_2\rangle$$

Figure 2.3.: Concrete semantics for the extended λ-calculus

The 0CFA for this language is in Figure 2.4. All of the rules from Figure 2.2 still apply except that some of the signatures have changed and the CALL_{out} rule changes to account for primitive operators.

The set of types is enriched by adding a fixed set of primitive types, e.g., *INT*, *PAIR*, etc., as well as for primitive procedures, e.g., `pair?`, `car`, etc., to the $\widehat{Val}$ lattice. The analysis splits abstract values into a function part and a non-function part to simplify the development of sub-0CFA in Section 2.4. Nevertheless, we notationally treat abstract values as sets. For example, $\{INT, PAIR, \lambda x.e\}$ is understood to mean $\langle\{INT, PAIR\}, \{\lambda x.e\}\rangle$.

Now that the analysis has primitive functions in addition to lambda functions, the CALL_{out} rule is refactored. It uses the *RET* function to determine the return type of a function. When the function is a lambda term, the result is exactly the same as before, but now the CALL_{out} rule also handles primitive functions that are not lambda terms. For example, the return value of `car` is $\top$, the top element of the $\widehat{Val}$ lattice.

The CONST rule handles constants by using the *ABS* function, which projects from concrete constants to abstract values.

Sequencing, i.e., $e_0; e_1$, is handled by the SEQ_{in}, SEQ_{mid} and SEQ_{out} rules. SEQ_{in} makes the first expression of a sequence reachable if the sequence is reachable. SEQ_{mid} makes the second expression of a sequence reachable if the first expression of the sequence returns. Finally, SEQ_{out} uses the result of the second expression of the sequence as the result of the entire sequence.

Conditionals, i.e., *if* e_0 e_1 e_2, are handled by the IF_{in}, IF_{mid} and IF_{out} rules. As with sequencing, IF_{in} makes the test of a conditional reachable if the entire conditional is reachable. Similar to SEQ_{mid}, the IF_{mid} rule makes the consequent and alternative reachable if the test returns either true or false values respectively. It uses $\top_t$ and $\top_f$, which represent the sets of all true and all false values respectively. Finally, in IF_{out}, the result of the conditional is obtained from the join of the results of the consequent and alternative.

Contexts: $C \in Ctxt = \ldots \mid (\Box; e_1) \mid (e_0; \Box) \mid (\mathit{if}\ \Box\ e_1\ e_2) \mid (\mathit{if}\ e_0\ \Box\ e_2) \mid (\mathit{if}\ e_0\ e_1\ \Box)$

Signatures:
$$\hat{v} \in \widehat{Val} = \widehat{Fun} \times \wp(\widehat{Tag}) \qquad \hat{f} \in \widehat{Fun} = \wp(\widehat{Lam} + \widehat{Prim})$$
$$\hat{t} \in \widehat{Tag} = \{FALSE, TRUE, INT, FLOAT, PAIR, \ldots\}$$
$$\widehat{Lam} = \{\lambda x_1.e_1, \lambda x_2.e_2, \lambda x_3.e_3, \ldots\}$$
$$o \in \widehat{Prim} = \{\texttt{pair?}, \texttt{car}, \texttt{cdr}, \ldots\}$$
$$\top_t = \widehat{Val} \backslash \{FALSE\} \qquad \top_f = \{FALSE\}$$
$$ABS \in Const \to \widehat{Val}$$
$$ABS(\texttt{\#f}) = \{FALSE\}$$
$$ABS(n) = \{INT\}$$
$$\ldots$$
$$RET \in \widehat{Val} \to \widehat{Val}$$
$$RET\,(\lambda x.e) = [\![e]\!]_{out}$$
$$RET\,(\texttt{car}) = \top$$
$$RET\,(\texttt{pair?}) = \{FALSE, TRUE\}$$
$$\ldots$$

$$\frac{[\![e_0]\!]_{out} \sqsupseteq \{\hat{f}\} \qquad [\![e_1]\!]_{out} \sqsupseteq \{\hat{v}_1\}}{[\![e_0\ e_1]\!]_{out} \sqsupseteq RET(\hat{f})}\ \text{CALL}_{out}$$

$$\frac{[\![c]\!]_{in} \sqsupseteq \top}{[\![c]\!]_{out} \sqsupseteq ABS(c)}\ \text{CONST}$$

$$\frac{[\![e_0; e_1]\!]_{in} \sqsupseteq \hat{r}}{[\![e_0]\!]_{in} \sqsupseteq \hat{r}}\ \text{SEQ}_{in}$$

$$\frac{[\![e_0]\!]_{out} \sqsupseteq \{\hat{v}_0\} \qquad \mathbb{K}(e_0) = (\Box; e_1)}{[\![e_1]\!]_{in} \sqsupseteq \top}\ \text{SEQ}_{mid}$$

$$\frac{}{[\![e_0; e_1]\!]_{out} \sqsupseteq [\![e_1]\!]_{out}}\ \text{SEQ}_{out}$$

$$\frac{[\![\mathit{if}\ e_0\ e_1\ e_2]\!]_{in} \sqsupseteq \hat{r}}{[\![e_0]\!]_{in} \sqsupseteq \hat{r}}\ \text{IF}_{in}$$

$$\frac{[\![e_0]\!]_{out} \sqsupseteq \hat{v}_0 \qquad \mathbb{K}(e_0) = (\mathit{if}\ \Box\ e_1\ e_2)}{[\![e_1]\!]_{in} \sqsupseteq \top \text{ if } \hat{v}_0 \sqcap \top_t \neq \bot \qquad [\![e_2]\!]_{in} \sqsupseteq \top \text{ if } \hat{v}_0 \sqcap \top_f \neq \bot}\ \text{IF}_{mid}$$

$$\frac{}{[\![\mathit{if}\ e_0\ e_1\ e_2]\!]_{out} \sqsupseteq [\![e_1]\!]_{out} \sqcup [\![e_2]\!]_{out}}\ \text{IF}_{out}$$

Figure 2.4.: Constraint rules for extended λ-calculus

2.2. Flow-graph implementation of CFA

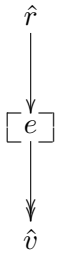
In order to solve the constraint rules for CFA efficiently, it is common to represent the problem as a flow graph [Jagannathan and Weeks, 1995, Heintze and McAllester, 1997b] with graph nodes denoting the flow variables $[\![e]\!]_{in}$ and $[\![e]\!]_{out}$ and directed edges denoting the flow of abstract values from one node to another. Each node in the flow graph has an associated value and an associated function. A node is *stable* if the value associated with it equals the result of applying the function associated with the node to the values associated with the nodes that have edges coming into the node. The goal of the flow graph is to find values to associate with each node such that every node in the graph is stable.

As a convention when we say that we compute a type or other phrasing to that effect, we often mean that we construct a flow-graph node that computes the type.

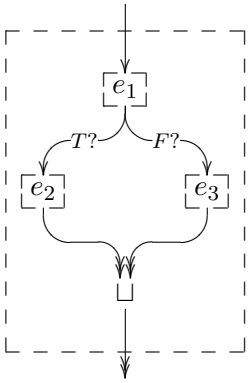
To compute a solution to a flow graph, we use the standard technique. We impose a finite lattice over the types of values associated with each node. Initially, we use $\perp$ for the value of each node. That is, we use the bottom element of the lattice for a particular node's initial type.

Then, a standard work-list algorithm iterates over the graph until convergence. Nodes in the work list may not be stable and may have stale values that need to be updated based on changed inputs to the node. Initially, all nodes are on the work list. A node is removed from the work list and a new value for its output is calculated based on the values of its inputs. If the output value changes, any nodes that connect to edges coming out of the current node are placed on the work list since they may not be stable. The algorithm continues selecting nodes form the work list and recalculating nodes until the graph converges and no unstable nodes remain.

For 0CFA augmented with reachability, an edge into an expression node models reachability $\hat{r}$, whereas an edge out of an expression node models possible result values, $\hat{v}$, as



(a) A general flow graph node



(b) A composite flow graph

Figure 2.5.: Flow graphs in 0CFA

depicted in Figure 2.5a. Thus, to construct a flow graph for the rules in Figure 2.2 or Figure 2.4, the analysis constructs a node for each $[\![e]\!]_{in}$ and each $[\![e]\!]_{out}$. The constraint rules then dictate the function for computing the output from the inputs. Thus, if the flow graph is stable, the constraint rules are satisfied. The solution thus found is also the minimal solution to the constraint rules, but that proof is omitted here. The interested reader is referred to Jagannathan and Weeks [1995].

For example, the flow graph for the expression `(if` e_1 e_2 e_3`)` contains nodes for the reachability and result values of the `if`, e_1, e_2, and e_3 expressions. This is depicted schematically in Figure 2.5b. The expressions e_1, e_2, and e_3 are drawn in outline to indicate they may contain other nodes internal to those expressions. The expression e_1 is reachable if and only if the `if` is reachable. Thus, there is an edge from the input of the `if` to e_1. Likewise e_2 and e_3 are reachable if and only if e_1 outputs a true or false value respectively. Thus, there are edges from e_1 to e_2 and e_3 filtered by $T?$ and $F?$ which test if the output value contains true or false values respectively. Finally, the output node of the `if` computes the lattice join ($\sqcup$) of the output nodes of e_2 and e_3.

A crucial property is that, under appropriate circumstances, flow graphs quickly converge to a solution. If (1) the values that flow through a graph are members of a finite-height lattice, L, (2) the output value of each node moves only monotonically up the lattice, and (3) the output value of a node can be computed in constant time from the input values, then the lattice will converge in $O(|L|\,(|E| + |N|))$ time where $|L|$, $|E|$ and $|N|$ are the height of L, the number of edges, and the number of nodes in the graph. Thus, when determining the asymptotic performance of a flow graph, we are most directly concerned with the number of nodes created as this, along with the lattice height and the time to recompute a single node, determines the time it takes for the flow graph to converge.

For CFA, a minor modification has to be made to the usual flow-graph algorithm. The initial graph contains no edges between functions and call sites, so the algorithm adds new edges to the flow graph as it discovers connections between functions and call sites. This does not affect convergence, however, since the maximum number of edges is bounded, and edges are added but never removed.

In the worst case, the algorithm adds an edge between each of $O(n)$ call sites and each of $O(n)$ functions in a program of size n, resulting in a graph with $O(n^2)$ edges. 0CFA uses a lattice over $\wp(\widehat{Lam})$, which has a height equal to the number of functions in the program. The lattice has height $O(n)$ and the graph has size $O(n^2)$, so a naively implemented 0CFA takes $O(n^3)$ time to compute. Slightly faster techniques are known for computing 0CFA, but they still take $O(n^3/\log n)$ time [Melski and Reps, 2000, Chaudhuri, 2008, Midtgaard and Van Horn, 2009].

2.3. Top and escaped functions

If a CFA is operating on a program that contains free variables, such as variables imported from libraries outside the scope of the analysis, the analysis does not know anything about

the values of those variables. This is handled by adding a top, $\top$, element to the lattice. This $\top$ denotes an unknown function [Shivers, 1988] and is used for the value of free variables. It represents not only any function from outside the scope of the analysis but also includes any function inside the scope of the analysis. This is to say, it subsumes all other functions in $\widehat{Lam}$.

Likewise, if a function is assigned to a free variable or exported to a library outside the scope of the analysis, the function may be called in locations unknown to the analysis. That the analysis has lost track of all the places where the function flows is represented by marking the function as *escaped*.

Top values and escaped functions can cause more top values and escaped functions. First, if the function position of a function call is $\top$, the return value of the function call is $\top$, and the arguments escape, since they are passed to an unknown function. Second, if a function escapes, its formal parameters become $\top$, and its return value escapes, since it might flow to places outside the scope of the analysis. Finally, when a set of functions is joined with a $\top$ value, the result is $\top$, and since a $\top$ value does not explicitly mention the functions combined into it, those functions are marked as escaped.

This handling of $\top$ and escaped functions is standard and is assumed throughout the rest of this dissertation.

2.4. Sub-0CFA

0CFA is often extended upwards to 1CFA, 2CFA or the general kCFA, which compute separate abstract values for different calling contexts of an expression. However, even the simplest of these, 1CFA, is EXPTIME complete [Van Horn and Mairson, 2008]. Further, even without flow sensitivity, the best known 0CFA algorithm takes $O(n^3/\log n)$ time [Midtgaard and Van Horn, 2009]. This is unlikely to be improved given the known

equivalence between 0CFA and 2NPDA [Heintze and McAllester, 1997a, Aho et al., 1968]. Thus, since we are aiming for $O(n \log n)$ time, we do not start with 0CFA. Instead we start with sub-0CFA, which takes only $O(n)$ time [Ashley and Dybvig, 1998]. Sub-0CFA is based on the observation that it is always a sound, conservative approximation to replace an abstract value, $\hat{v}_1$, with any other abstract value, $\hat{v}_2$, provided that $\hat{v}_2$ approximates $\hat{v}_1$. That is to say $\hat{v}_2 \sqsupseteq \hat{v}_1$. The general form of this technique is called *widening*. Between the steps in the iteration of the flow graph, the abstract values on each node are examined and a widening operator is applied to these value. The widening operator selects the new abstract values that are then used as the new values for each flow-graph node. In theory, the widening operator can be any function so long as it only ever moves values up their corresponding lattice. The operator can even depend on correlations between values at different nodes. However, in practice we want to choose an operator that makes the flow graph converge quickly and that is fast to compute.

A particularly common and useful special case of this calculates the new value for each node independently of every other node. In this case, the formalism can be simplified by doing away with the widening operator and instead restricting abstract values to a sub-lattice of the power-set lattice of abstract values. This technique is easy to implement by choosing a data type for abstract values that has appropriate meet ($\sqcap$) and join ($\sqcup$) operators. It is also easier to predict the behavior of using a restricted lattice than using a widening operator since its results depend only on local effects.

The original sub-0CFA [Ashley and Dybvig, 1998] operates in terms of a widening operator that takes into account how many iterations the flow graph has gone through. However, for the reasons mentioned above we use a variant that merely restricts the lattice. It bounds both the size of the graph and the height of the lattice by approximating all non-singleton sets of functions with $\top$. That is to say, it uses a sub-lattice of the power-set lattice that contains only $\top$, $\bot$, and singleton sets. Any set of size two or more is approximated by

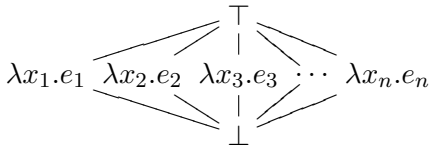


Figure 2.6.: Flattened lattice of functions

$\top$ while all other sets are represented by themselves. This conservative approximation of 0CFA's power-set lattice has a constant height and is shown in Figure 2.6. Whenever two or more different functions are joined by $\sqcup$, the result is $\top$. As a result, the values that flow to the function position of a particular call site either contain at most one function or are approximated by $\top$ and thus add at most a linear number of edges to the graph.

This approach lets more functions escape than in 0CFA, but this is not as bad as it might seem. Flowing to two different places does *not* cause a function to escape. Functions escape only when two or more flow to the same point, i.e., when a call site performs some sort of dispatch. For example, when running the analysis over the following, both `f` and `g` escape, since they both flow into `fg`, but `h` is not affected.

```
(let ([f (lambda (x) e1)]                                          (2.1)
      [g (lambda (y) e2)]
      [h (lambda (z) e3)])
  (let ([fg (if (read) f g)]) (f (g (fg (h (h e4)))))))
```

In the general case, Ashley and Dybvig [1998] define sub-0CFA by a projection or widening operator. When this operator restricts sets of values to either singletons or $\top$, it is equivalent to what is done in Figure 2.6. Other projection operators produce lattices that can be of constant or even logarithmic height and result in linear or nearly linear analyses. For example, instead of sets of at most one function, the projection may limit sets to at most k functions for some constant k.

2.5. Non-function types

Programming languages usually have more values than just functions. Thus, we add a fixed set of primitive types, e.g., INT, $PAIR$, etc., to the $\widehat{Val}$ lattice. However, abstract values are split into a function part and a non-function part. The function part operates over the same flattened lattice as sub-0CFA, but since there are a fixed number of non-function types we allow the non-function part to operate over the full power-set of non-function types. Nevertheless, we notationally treat abstract values as sets. For example, $\{INT, PAIR, \lambda x.e\}$ is understood to mean $\langle\{INT, PAIR\}, \{\lambda x.e\}\rangle$.

3. Flow Sensitivity

The control-flow analyses described in Chapter 2 are flow insensitive. This means all references to a variable are treated as having the same value as the binding site of the variable. Consider these examples from the introduction:

```
(let ([x (cons e1 e2)]) (car x))                    (3.1)
(let ([x (read)]) (if (pair? x) (car x) #f))        (3.2)
(let ([x (read)]) (begin (cdr x) (car x)))          (3.3)
```

With a flow-insensitive analysis, the reference to `x` in Example 3.1 is known to be a pair but, in the Example 3.2 and Example 3.3, all references to `x` are treated as $\top$.

Type information can be gained, however, from the explicit and implicit dynamic type checks in the second and third expressions. In Example 3.2, we can deduce from the explicit pair check, `(pair? x)`, that `x` must be a pair when `car` is called. In Example 3.3, we can also deduce that `x` must be a pair when `car` is called, since the implicit pair check in `cdr` guarantees that it returns only if its argument is a pair.

In these cases, the monotonicity requirement for implementing the analysis in a flow graph is still satisfied since monotonicity requires only that individual nodes in the flow graph change monotonically. In Example 3.3, every node in the flow graph starts with $\bot$ a value. As the flow graph converges to a solution, the value for `x` before `(cdr x)` moves up the lattice to $\top$. The value for `x` after `(cdr x)`, however, moves up only to pair values. Both points in the program move monotonically up the lattice, even though a lower value in the lattice occurs after a higher value in the evaluation order of the code.

Information is *constructive* when it is learned from operations that are constructing values as with Example 3.1. Information is *observational* when it is learned from operations that are observing values as with Example 3.2 and Example 3.3.

Observational information is *restrictive*, since it restricts the type of a variable or value, as in the restriction of `x` to the pair type in Example 3.2 and Example 3.3. If observational information restricts a type to two or more disjoint types, the type is $\bot$. In general, wherever a $\bot$ type occurs, the sequentially following code is unreachable, i.e., dead, and can be discarded.

Without this sort of observational information, a value that becomes $\top$ will stay at $\top$. These $\top$ values arise in a number of ways and as they represent no knowledge they are unoptimizable. Thus, it is particularly important that the analysis be able to move away from $\top$ to more restricted values. For example, the return value of a function like `car` or `cdr` is $\top$. Also, whenever two functions flow to the same location they are approximated by $\top$. Finally, the arguments to any top-level-bound function that is callable from outside the module being analyzed are approximated by $\top$. Each of these sources of $\top$ can in turn cause further values to be approximated by $\top$ as the $\top$ value joins with other values or forces procedures to be marked as escaped and thus have arguments approximated by $\top$.

To collect observational information, the analysis must be flow-sensitive as the type information about a variable is different at different points, e.g., before and after an observation. This chapter presents such an analysis in two stages. First, it presents an analysis that is flow sensitive and gathers observational information only from functions like `car` that unconditionally restrict the type of their argument. The approach to this form of flow sensitivity is standard. It then generalizes this and presents an analysis that also gathers observational information from functions that restrict the type of their argument conditionally. For example, the argument of `pair?` is limited to pairs if and only if `pair?` returns true. This form of flow sensitivity is novel.

Expressions:
$$e \in Exp = x \mid \lambda x.e \mid e\ e \mid c \mid e; e \mid \mathit{if}\ e\ e\ e$$
$$c \in Const = \texttt{\#f} \mid n \mid \dots$$
$$n \in Num = 0 \mid 1 \mid 2 \mid \dots$$

Contexts:
$$C \in Ctxt = \Box \mid (\Box\ e_1) \mid (e_0\ \Box) \mid (\lambda x.\Box) \mid (\Box; e_1) \mid (e_0; \Box) \mid (\mathit{if}\ \Box\ e_1\ e_2) \mid (\mathit{if}\ e_0\ \Box\ e_2) \mid (\mathit{if}\ e_0\ e_1\ \Box)$$

Signatures:
$$\mathbb{K} \in Exp \to Ctxt$$
$$\hat{r} \in Bool = \{\bot, \top\}$$
$$\hat{\rho} \in \widehat{Env} = Var \to \widehat{Val}$$
$$\hat{v} \in \widehat{Val} = \widehat{Fun} \times \wp(\widehat{Tag})$$
$$\hat{f} \in \widehat{Fun} = \wp(\widehat{Lam} + \widehat{Prim})$$
$$\hat{t} \in \widehat{Tag} = \{FALSE, TRUE, INT, FLOAT, PAIR, \dots\}$$
$$\widehat{Lam} = \{\lambda x_1.e_1, \lambda x_2.e_2, \lambda x_3.e_3, \dots\}$$
$$o \in \widehat{Prim} = \{\texttt{pair?}, \texttt{car}, \texttt{cdr}, \dots\}$$

$$ARG \in \widehat{Env} \times Exp \times \widehat{Val} \to \widehat{Env}$$
$$ARG(\hat{\rho}, e, \hat{v}) = \begin{cases} \bot & \text{if } \hat{v} = \bot \\ \hat{\rho} & \text{if } \hat{v} \neq \bot \wedge e \notin Var \\ \hat{\rho}[e \mapsto \hat{\rho}(e) \sqcap \hat{v}] & \text{if } e \in Var \end{cases}$$

Figure 3.1.: Common definitions

3.1. Flow-sensitivity for unconditional observers

To recover observational information from functions like `car`, an analysis must be flow sensitive.[1] A flow-insensitive analysis takes information about a variable's abstract value directly from its binding site to each reference. The information about a variable is the

[1] The term *flow sensitive* has some ambiguity to it. Even the coiner of the term, Banning [1979], used it with three different meanings [Marlowe et al., 1995]. This dissertation uses the glossary definition [Mogensen, 2000] and takes it to be a property that describes an analysis that takes the order and sequence of evaluation into account. This contrasts with analyses like the CFA in Chapter 2 that are flow insensitive and do not take evaluation order into account.

same at all references to the variable. To be flow-sensitive, the analysis adjusts the abstract value of variables as it traces the execution flow of the program. Consider Example 3.3 on page 19 which used `(cdr x)` and `(car x)`. With flow-sensitivity, the variable `x` starts at its binding site with the abstract value $\top$. It then flows to `(cdr x)`. On entry to `(cdr x)`, `x` still has the abstract value $\top$. Since `cdr` throws an error and does not return unless its argument is a pair, the analysis learns that `x` is a pair on exit from `(cdr x)`. This then flows to `(car x)`. Thus, `(car x)` is only ever called with a pair as argument. The implicit pair check in `cdr` prevents non-pair values from flowing to the `car`, so the implicit pair check in the `car` can be safely omitted.

To gather restrictive information, each function is annotated with the abstract value that each argument must be for the function to return. For example, if `car` returns, its argument must be a pair. This is not limited to built-in primitives. For user-defined functions, the analysis examines the abstract value of each formal parameter after flowing through the function body. In the following example, if `g` returns, the analysis knows the function's argument, `y`, is a pair. Thus, `x` must be a pair after returning from `(g x)`. Consequently, the implicit pair check in `cdr` is redundant and can be safely omitted.

```
(let ([x (read)]                                                    (3.4)
      [g (lambda (y) (+ (car y) 1))])
  (g x)
  (cdr x))
```

The formal semantics for an analysis with flow sensitivity for unconditional observers is a straightforward extension of the analysis in Figure 2.4 for the flow-insensitive 0CFA and is shown in Figure 3.2 and Figure 3.3. It includes sequencing by threading the environment through the execution flow of the program. Each expression still has an associated reachability flag and result value, but now each expression also has two environments associated with it. One tracks the types of variables when entering the expression. The other tracks the types of variables when exiting the expression. At each function call, arguments are

restricted to only those abstract values that are compatible with the particular function returning. After `(car x)` for example, `x` is restricted to pairs. Both the entering and exiting environments are treated as reduced abstract domains [Cousot and Cousot, 1979] thus equating abstract elements with the same meaning, i.e., concretization. Hence, if any component of an environment is $\bot$, then all components of the environment are forced to be $\bot$. For example, if `x` is known to be an integer, then `x` is $\bot$ after `(car x)`. This causes all components of the exiting environment to be $\bot$. In addition, as part of the reduced abstract domain, the return value of `(car x)` becomes $\bot$. This models the fact that `(car x)` does not return if `x` is an integer.

This form of flow sensitivity will not learn information from some observers hidden inside function calls. For example, in the following, it will not learn that `x` is a pair at e_2 even though it will lean that `x` is a pair at e_1.

```
(let ([x read])                                                    (3.5)
  (let ([f (lambda () (car x) e1)])
    (f)
    e2))
```

However, when a variable is immutable, ignoring restrictive information about that variable is sound. The worst that happens is that the analysis conservatively over approximates some values.

On the other hand, when a variable is mutated, it is not a safe approximation to ignore the mutation, since the value might become a value not approximated by the original value. For example, in Example 3.6, ignoring the `set!` inside `f` would lead to the conclusion that `x` is a pair at the call to `car`.

```
(let ([x (cons 1 2)])                                              (3.6)
  (let ([f (lambda () (set! x 3))])
    (f)
    (car x)))
```

For simplicity of presentation, the discussion of how to handle this is deferred until Section 6.7. For now all variables are assumed to be immutable.

Signatures:

$$
\begin{aligned}
[\![e]\!]_{in} &\in Bool \times \widehat{Env} \\
[\![e]\!]_{out} &\in \widehat{Val} \times \widehat{Env} \\
RET &\in \widehat{Val} \to \widehat{Val} \times \widehat{Env} \\
RET\left(\hat{f}\right) &= \bigsqcup_{f \in \hat{f}} RET\,(f) \\
RET\,(\lambda x.e) &= \langle \hat{v},\, \hat{\rho}(x) \rangle \quad \text{where } \langle \hat{v},\, \hat{\rho} \rangle = [\![e]\!]_{out} \\
RET\,(\texttt{car}) &= \langle \top,\, \{PAIR\} \rangle \\
RET\,(\texttt{pair?}) &= \langle \{FALSE,\, TRUE\},\, \top \rangle \\
&\cdots
\end{aligned}
$$

Figure 3.2.: Definitions for unconditional observers

This simple form of flow sensitivity handles functions like `car` that unconditionally provide observational information about their arguments when they return. However, it fails to handle predicates such as `pair?`. The fact that a call to `pair?` returns says nothing about its argument. Rather, the information is conditional. Whether it returns a true or false value tells the analysis whether the argument is a pair.

3.2. Flow-sensitivity for conditional observers

To handle conditional or predicated observers, we generalize the environments flowing through the program and make the analysis *predicate aware*. For the exit of an expression, the analysis stores one environment for when the expression returns a true value and another for when the expression returns a false value. The environment stored for entry to the expression remains the same as before.

Predicate awareness is important when recovering types in a dynamically typed language as predicates are the most primitive way for type information to be learned. For example, `x` is a pair if and only if `(pair? x)` is true. Type information can be learned from non-

$$\frac{[\![x]\!]_{in} \sqsupseteq \langle \top, \hat{\rho} \rangle}{[\![x]\!]_{out} \sqsupseteq \langle \hat{\rho}(x), \hat{\rho}[x \mapsto \hat{\rho}(x)] \rangle} \text{ VAR}$$

$$\frac{[\![\lambda x.e]\!]_{in} \sqsupseteq \langle \top, \hat{\rho} \rangle}{[\![\lambda x.e]\!]_{out} \sqsupseteq \langle \{\lambda x.e\}, \hat{\rho} \rangle} \text{ LAMBDA}$$

$$\frac{[\![e_0\ e_1]\!]_{in} \sqsupseteq \langle \hat{r}, \hat{\rho} \rangle}{[\![e_0]\!]_{in} \sqsupseteq \langle \hat{r}, \hat{\rho} \rangle} \text{ CALL}_{in}$$

$$\frac{[\![e_0]\!]_{out} \sqsupseteq \langle \hat{v}_0, \hat{\rho} \rangle \qquad \mathbb{K}(e_0) = (\square\ e_1)}{[\![e_1]\!]_{in} \sqsupseteq \langle \top, \hat{\rho} \rangle} \text{ CALL}_{mid}$$

$$\frac{\begin{array}{c}[\![\lambda x.e_\lambda]\!]_{in} \sqsupseteq \langle \top, \hat{\rho}_\lambda \rangle \qquad [\![e_0]\!]_{out} \sqsupseteq \langle \{\lambda x.e_\lambda\}, \hat{\rho}^{e_0} \rangle \\ [\![e_1]\!]_{out} \sqsupseteq \langle \hat{v}_1, \hat{\rho}^{e_1} \rangle \qquad \mathbb{K}(e_1) = (e_0\ \square)\end{array}}{[\![e_\lambda]\!]_{in} \sqsupseteq \langle \top, \hat{\rho}_\lambda \rangle} \text{ CALL}_{fun}$$

$$\frac{\begin{array}{c}[\![e_0]\!]_{out} \sqsupseteq \left\langle \{\hat{f}\}, \hat{\rho}^{e_0} \right\rangle \qquad [\![e_1]\!]_{out} \sqsupseteq \langle \hat{v}_1, \hat{\rho}^{e_1} \rangle \\ \langle \hat{v}, \hat{v}_{arg} \rangle = RET(\hat{f}) \qquad \hat{\rho}'_{arg} = ARG(\hat{\rho}^{e_1}, e_1, \hat{v}_1 \sqcap \hat{v}_{arg})\end{array}}{[\![e_0\ e_1]\!]_{out} \sqsupseteq \left\langle \hat{v}, \hat{\rho}'_{arg} \right\rangle} \text{ CALL}_{out}$$

$$\frac{[\![c]\!]_{in} \sqsupseteq \langle \top, \hat{\rho} \rangle}{[\![c]\!]_{out} \sqsupseteq \langle ABS(c), \hat{\rho} \rangle} \text{ CONST}$$

$$\frac{[\![e_0; e_1]\!]_{in} \sqsupseteq \langle \hat{r}, \hat{\rho} \rangle}{[\![e_0]\!]_{in} \sqsupseteq \langle \hat{r}, \hat{\rho} \rangle} \text{ SEQ}_{in}$$

$$\frac{[\![e_0]\!]_{out} \sqsupseteq \langle \hat{v}_0, \hat{\rho} \rangle \qquad \mathbb{K}(e_0) = (\square; e_1)}{[\![e_1]\!]_{in} \sqsupseteq \langle \top, \hat{\rho} \rangle} \text{ SEQ}_{mid}$$

$$\frac{}{[\![e_0; e_1]\!]_{out} \sqsupseteq [\![e_1]\!]_{out}} \text{ SEQ}_{out}$$

$$\frac{[\![if\ e_0\ e_1\ e_2]\!]_{in} \sqsupseteq \langle \hat{r}, \hat{\rho} \rangle}{[\![e_0]\!]_{in} \sqsupseteq \langle \hat{r}, \hat{\rho} \rangle} \text{ IF}_{in}$$

$$\frac{[\![e_0]\!]_{out} \sqsupseteq \langle \hat{v}_0, \hat{\rho} \rangle \qquad \mathbb{K}(e_0) = (if\ \square\ e_1\ e_2)}{[\![e_1]\!]_{in} \sqsupseteq \langle \top, \hat{\rho} \rangle \qquad [\![e_2]\!]_{in} \sqsupseteq \langle \top, \hat{\rho} \rangle} \text{ IF}_{mid}$$

$$\frac{}{[\![if\ e_0\ e_1\ e_2]\!]_{out} \sqsupseteq [\![e_1]\!]_{out} \sqcup [\![e_2]\!]_{out}} \text{ IF}_{out}$$

Figure 3.3.: Rules for unconditional observers

Signatures:

$$\begin{aligned}
[\![e]\!]_{in} &\in Bool \times \widehat{Env} \\
[\![e]\!]_{out} &\in \widehat{Val} \times \widehat{Env} \times \widehat{Env} \\
RET\left(\hat{f}\right) &= \bigsqcup_{f \in \hat{f}} RET\,(f) \\
RET\,(\lambda x.e) &= \langle \hat{v},\, \hat{\rho}_t(x),\, \hat{\rho}_f(x) \rangle\, \langle \hat{v},\, \hat{\rho}(x) \rangle \quad \text{where } \langle \hat{v},\, \hat{\rho}_t,\, \hat{\rho}_f \rangle = [\![e]\!]_{out} \\
RET\,(\texttt{car}) &= \langle \top,\, \{PAIR\},\, \{PAIR\} \rangle \\
RET\,(\texttt{pair?}) &= \langle \{FALSE,\, TRUE\},\, \{PAIR\},\, \top \backslash \{PAIR\} \rangle \\
&\cdots
\end{aligned}$$

Figure 3.4.: Predicate aware definitions

conditional observers such as `car`, but handling predicates is both more general and more effective. Predicate awareness is more general because functions like `car` can be treated as degenerate predicates. For example, with `car`, both true and false return values indicate that its argument is a pair. Predicate awareness is also more effective since it allows the analysis to take advantage of dynamic type checks that are in the code explicitly. For example, if the code is written in the style of Example 3.7 and contains explicit error checking code, the analysis can use the type information learned from `correct-types?` to eliminate type checks in e_1.

```
(lambda (x1 x2 ··· xn)                                          (3.7)
  (if (correct-types? x1 x2 ··· xn) e1 e2))
```

Another common case is when a recursive function uses a dynamic type check to determine when to stop recurring. For example, a function that recurs on a list might be structured in the following style.

```
(letrec ([f (lambda (x)                                         (3.8)
              (if (pair? x) e1 e2))])
  e3)
```

In this case, learning type information from the `pair?` predicate allows the analysis to eliminate implicit type checks of `x` in e_1.

$$\frac{[\![x]\!]_{in} \sqsupseteq \langle \top, \hat{\rho} \rangle}{[\![x]\!]_{out} \sqsupseteq \langle \hat{\rho}(x), \hat{\rho}[x \mapsto \hat{\rho}(x) \sqcap \top_t], \hat{\rho}[x \mapsto \hat{\rho}(x) \sqcap \top_f] \rangle} \text{ VAR}$$

$$\frac{[\![\lambda x.e]\!]_{in} \sqsupseteq \langle \top, \hat{\rho} \rangle}{[\![\lambda x.e]\!]_{out} \sqsupseteq \langle \{\lambda x.e\}, \hat{\rho}, \hat{\rho} \rangle} \text{ LAMBDA}$$

$$\frac{[\![e_0\ e_1]\!]_{in} \sqsupseteq \langle \hat{r}, \hat{\rho} \rangle}{[\![e_0]\!]_{in} \sqsupseteq \langle \hat{r}, \hat{\rho} \rangle} \text{ CALL}_{in}$$

$$\frac{[\![e_0]\!]_{out} \sqsupseteq \langle \hat{v}_0, \hat{\rho}_t, \hat{\rho}_f \rangle \qquad \mathbb{K}(e_0) = (\square\ e_1)}{[\![e_1]\!]_{in} \sqsupseteq \langle \top, \hat{\rho}_t \sqcup \hat{\rho}_f \rangle} \text{ CALL}_{mid}$$

$$\frac{\begin{array}{c}[\![\lambda x.e_\lambda]\!]_{in} \sqsupseteq \langle \top, \hat{\rho}_\lambda \rangle \qquad [\![e_0]\!]_{out} \sqsupseteq \left\langle \{\lambda x.e_\lambda\}, \hat{\rho}_t^{e_0}, \hat{\rho}_f^{e_0} \right\rangle \\ [\![e_1]\!]_{out} \sqsupseteq \left\langle \hat{v}_1, \hat{\rho}_t^{e_1}, \hat{\rho}_f^{e_1} \right\rangle \qquad \mathbb{K}(e_1) = (e_0\ \square)\end{array}}{[\![e_\lambda]\!]_{in} \sqsupseteq \langle \top, \hat{\rho}_\lambda \rangle} \text{ CALL}_{fun}$$

$$\frac{\begin{array}{c}[\![e_0]\!]_{out} \sqsupseteq \left\langle \{\hat{f}\}, \hat{\rho}_t^{e_0}, \hat{\rho}_f^{e_0} \right\rangle \quad [\![e_1]\!]_{out} \sqsupseteq \left\langle \hat{v}_1, \hat{\rho}_t^{e_1}, \hat{\rho}_f^{e_1} \right\rangle \\ \langle \hat{v}, \hat{v}_t, \hat{v}_f \rangle = RET(\hat{f}) \quad \forall i \in \{t, f\}.\hat{\rho}'_i = ARG(\hat{\rho}_t^{e_1} \sqcup \hat{\rho}_f^{e_1}, e_1, \hat{v}_1 \sqcap \hat{v}_i)\end{array}}{[\![e_0\ e_1]\!]_{out} \sqsupseteq \left\langle \hat{v}, \hat{\rho}'_t, \hat{\rho}'_f \right\rangle} \text{ CALL}_{out}$$

$$\frac{[\![c]\!]_{in} \sqsupseteq \langle \top, \hat{\rho} \rangle}{[\![c]\!]_{out} \sqsupseteq \langle ABS(c), \hat{\rho}, \hat{\rho} \rangle} \text{ CONST}$$

$$\frac{[\![e_0; e_1]\!]_{in} \sqsupseteq \langle \hat{r}, \hat{\rho} \rangle}{[\![e_0]\!]_{in} \sqsupseteq \langle \hat{r}, \hat{\rho} \rangle} \text{ SEQ}_{in}$$

$$\frac{[\![e_0]\!]_{out} \sqsupseteq \langle \hat{v}_0, \hat{\rho}_t, \hat{\rho}_f \rangle \qquad \mathbb{K}(e_0) = (\square; e_1)}{[\![e_1]\!]_{in} \sqsupseteq \langle \top, \hat{\rho}_t \sqcup \hat{\rho}_f \rangle} \text{ SEQ}_{mid}$$

$$\frac{}{[\![e_0; e_1]\!]_{out} \sqsupseteq [\![e_1]\!]_{out}} \text{ SEQ}_{out}$$

$$\frac{[\![if\ e_0\ e_1\ e_2]\!]_{in} \sqsupseteq \langle \hat{r}, \hat{\rho} \rangle}{[\![e_0]\!]_{in} \sqsupseteq \langle \hat{r}, \hat{\rho} \rangle} \text{ IF}_{in}$$

$$\frac{[\![e_0]\!]_{out} \sqsupseteq \langle \hat{v}_0, \hat{\rho}_t, \hat{\rho}_f \rangle \qquad \mathbb{K}(e_0) = (if\ \square\ e_1\ e_2)}{[\![e_1]\!]_{in} \sqsupseteq \langle \top, \hat{\rho}_t \rangle \qquad [\![e_2]\!]_{in} \sqsupseteq \langle \top, \hat{\rho}_f \rangle} \text{ IF}_{mid}$$

$$\frac{}{[\![if\ e_0\ e_1\ e_2]\!]_{out} \sqsupseteq [\![e_1]\!]_{out} \sqcup [\![e_2]\!]_{out}} \text{ IF}_{out}$$

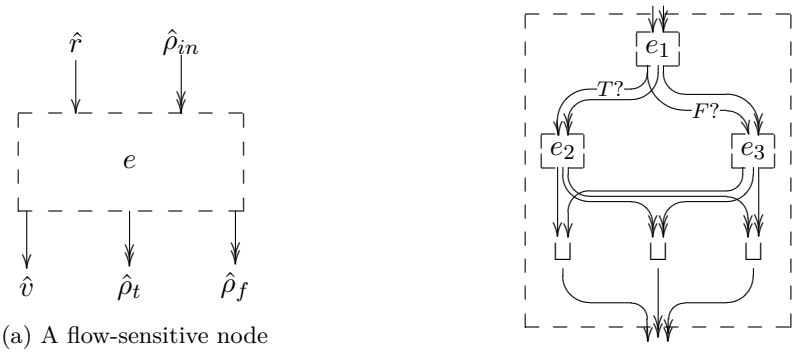
Figure 3.5.: Predicate aware rules

Though we use simple type predicates like `pair?` as a prototypical example, these techniques generalize to other properties that can be tested by a predicate. Examples include the numerical range of a variable, whether an index is within a vector's bound, or whether a list is sorted.

Figure 3.4 and Figure 3.5 present this formally. They include two environments in $[\![e]\!]_{out}$. One contains abstract values for when e returns true and the other for when e returns false. For example, $[\![(pair?\ x)]\!]_{out}$ has `x` as a pair in the true environment and as a non-pair in the false environment. These true and false environments are used by the IF_{mid} rule. The true environment of the test flows to the entering environment of the true branch. The false environment of the test flows to the entering environment of the false branch.

In the abstract semantics of Figure 3.5, gathering restrictive information from a function call, e.g., `(car x)` or `(pair? x)`, is implemented by the CALL_{out} rule. First, the values and environments that flow out of e_0 and e_1 are collected. Next, *RET* examines any functions, $\hat{f}$, flowing out of e_0 and returns three abstract values. One value is the return value of $\hat{f}$. The other two are the values that the argument to $\hat{f}$ must be for $\hat{f}$ to return either true or false respectively. Finally, *ARG* uses this information to determine for both the true and false cases if the function could return to this call site given the abstract value of the call site's argument. For example, `(car 3)` never returns. If the argument is a variable, *ARG* restricts the variable to the appropriate abstract value. Thus, after `(pair? x)`, `x` is restricted to pairs and non-pairs in the true and false cases respectively.

Each primitive or function has one abstract value for its argument for when it returns true and another for when it returns false. For example, with `pair?`, *RET* returns the abstract value for pairs in the true case and the abstract value for non-pairs in the false case. Unconditional observers like `car` have the same abstract value in both the true and false cases. For user-defined functions, the same information is obtained from the exiting true and false environments of the body of the function.



(a) A flow-sensitive node

(b) A composite flow graph

Figure 3.6.: Flow graphs for flow-sensitive 0CFA

3.3. Flow-graph representation of flow-sensitivity

As with flow-insensitive CFA, these constraint rules can be implemented by a flow graph. Yet, while information can flow directly from variable bindings to variable references in flow-insensitive CFA, this is not sufficient in flow-sensitive CFA. Instead, the abstract value for a variable must be threaded through each expression. In addition to the reachability flags and result abstract values, the analysis associates an environment with each entry edge of an expression and a true and a false environment with each exit edge of an expression. This is depicted in Figure 3.6a. The lines with single-headed arrows represent the flow of single values, and lines with double-headed arrows represent the flow of environments. Figure 3.6b shows how to extend the composite flow graph for `if` from Figure 2.5b to account for the extra edges and illustrates the flow of the true and false environments.

This graph formulation still has only linearly many edges, but some edges now contain environments. The lattice of an environment is the Cartesian product of the lattice for each variable, so the lattice height of an environment is linear in the number of variables. Thus, the flow graph has a linear number of edges with linear height lattices and consequently takes quadratic time to converge in the worst case.

In fact, this quadratic behavior is quite easy to trigger. Consider the following expression.

```
(lambda (x1 x2 ··· xn)
  (car x1) (car x2) ··· (car xn)
  (cdr x1) (cdr x2) ··· (cdr xn))
```

This expression has size $O(n)$ and n variables in scope. Thus, the environments that flow from a `car` or `cdr` to the following expression each contain n variables. Since there are $O(n)$ places where environments flow between expressions, the flow graph for this expression using the constraint rules as given in Figure 3.5 takes quadratic time to converge.

4. Fast Flow Sensitivity

The flow-graph based algorithm for flow-sensitive CFA described in Chapter 3 is quadratic because environments are threaded through each expression. Contrast this with the flow-insensitive sub-0CFA from Chapter 2 where the abstract value of a variable flows directly from its binding location to each reference without threading through intervening expressions. To implement flow sensitivity efficiently, we adopt a similar approach. Instead of flowing abstract values from a variable binding site to each reference, the analysis flows the information directly from one occurrence of the variable to the next, following the control flow of the program. The abstract value of a variable is adjusted at each occurrence.

When moving from one occurrence to the next, the true and false values for each variable may join or swap with each other. For example, consider how abstract values for `x` flow from `(pair? x)` to `(car x)` in the following expression. It is of the sort that may arise via the expansion of boolean connectives such as `and`, `or`, and `not`. Assume `x` is not referenced in e_1 or e_2.

$$\texttt{(if (if (pair? x) } e_1 \ e_2\texttt{) (car x) } e_3\texttt{)} \tag{4.1}$$

What the flow-sensitive analysis learns about `x` at `(car x)` depends on the return values of e_1 and e_2. If e_1 evaluates to only true values and e_2 evaluates to only false values, then `x` at `(car x)` is always a pair. If, however, e_1 evaluates to only false values and e_2 evaluates to only true values, then `x` at `(car x)` is never a pair. Other cases arise if either expression diverges or returns both true and false values.

Likewise, consider the following expression where restrictive information is learned from `(car x)` in one of the branches of the inner `if`. As before, assume `x` is not referenced in expressions e_0, e_1, or e_2.

```
(let ([x (read)])                                                    (4.2)
  (if (if e0 (begin (car x) e1) e2)
    (cdr x)
    e3))
```

After `(car x)`, `x` is known to be a pair, but the abstract value of `x` at `(cdr x)` depends upon the return values of e_1 and e_2. If e_2 returns a true value, then `(cdr x)` is reachable without passing through `(car x)`. If e_2 evaluates to only false values, then all paths to `(cdr x)` go through `(car x)`, and `x` at `(cdr x)` must be a pair. Interestingly, the return value of e_0 is not needed to determine this, since if e_2 always evaluates to a false value, then `(cdr x)` is reachable only when e_0 is a true value and sends control through `(car x)`.

Dealing with how the abstract values of variables change as they flow through the program while taking only $O(n \log n)$ time is the primary technical challenge of this analysis. The remainder of this chapter shows how to do this. The result is an analysis that takes only linear-log time and produces the same results as the analysis in Chapter 3.

First, Section 4.1 defines a skipping function $\mathcal{V}_{C,e}$ that allows values to move from one point in the program to another without threading through each intervening expression, while accounting for changes that happen as they flow through each expression. Next, Section 4.3 explains how to determine where to use the skipping functions versus the constraint rules. Then Section 4.4 describes the data structures used to cache the skipping functions efficiently so that each one can be computed or updated in logarithmic time. Since the algorithm in Section 4.3 ensures that only a linear number of skipping functions are used, the entire analysis then takes only linear-log time. Finally, Section 4.5 puts all of these together into a linear-log time algorithm that computes results identical to those of the less efficient algorithm.

$$\mathcal{G}_t(e,\ u) = (\hat{v} \sqcap \top_t) \neq \bot\ ?\ u\ :\ \bot$$
$$\mathcal{G}_f(e,\ u) = (\hat{v} \sqcap \top_f) \neq \bot\ ?\ u\ :\ \bot$$
$$\text{where } \langle \hat{v},\ \hat{\rho}_t,\ \hat{\rho}_f \rangle = [\![e]\!]_{out}$$

Figure 4.1.: True and false expression guards

4.1. Context skipping

In Example 4.1 with `pair?`, the traditional algorithm first flows the abstract value of `x` from the exiting environments of `(pair? x)` into the entering environments of e_1 and e_2, then through e_1 and e_2, and finally from the exiting environments of e_1 and e_2 out of both `if` expressions and into `(car x)`. When flowing through e_1 and e_2, the abstract value of `x` is threaded through every subexpression of e_1 and e_2. If `x` is not referenced in e_1 and e_2, however, these expressions can be skipped. Intuitively, the true and false environments that contain `x` might be swapped or joined, but the value of `x` in each environment does not fundamentally change. The following theorem reflects this intuition.

Theorem 4.1. *Expression Skipping*

If `x` *is a variable not mentioned in* e *then*

$$\hat{\rho}_t(x) = \mathcal{G}_t(e,\ \hat{\rho}_{in}(x)) \quad \textit{and}$$
$$\hat{\rho}_f(x) = \mathcal{G}_f(e,\ \hat{\rho}_{in}(x))$$
$$\textit{where } \langle \hat{r},\ \hat{\rho}_{in} \rangle = [\![e]\!]_{in}$$
$$\langle \hat{v},\ \hat{\rho}_t,\ \hat{\rho}_v \rangle = [\![e]\!]_{out}$$
$$\textit{and } \mathcal{G}_t(e,\ u) \textit{ and } \mathcal{G}_f(e,\ u) \textit{ are as defined in Figure 4.1.}$$

Proof. The proof proceeds by induction on e while using the constraint rules in Figure 3.5 on page 27. Here only the proof of $\hat{\rho}_t(x) = \mathcal{G}_t(e,\ \hat{\rho}_{in}(x))$ is shown as the proof of $\hat{\rho}_f(x) =$

$\mathcal{G}_f(e,\ \hat{\rho}_{in}(x))$ is almost identical. As a notational convenience in this and subsequent proofs, $\langle \hat{r}^e,\ \hat{\rho}^e_{in} \rangle = [\![e]\!]_{in}$ and $\left\langle \hat{v}^e,\ \hat{\rho}^e_t,\ \hat{\rho}^e_f \right\rangle = [\![e]\!]_{out}$ for any given e.

Case 1. If $\hat{v} \sqcap \top_t = \bot$, then $\mathcal{G}_t(e,\ \hat{\rho}_{in}(x)) = \bot$. Thus, by the reduced abstract domain, we have $\hat{\rho}_t(x) = \bot$. Hence, $\mathcal{G}_t(e,\ \hat{\rho}_{in}(x)) = \hat{\rho}_t(x)$.

Case 2. If $\hat{v} \sqcap \top_t \neq \bot$ and e is a variable $x' \neq x$, a lambda expression $\lambda x'.e$, or a constant c, then by the VAR, LAMBDA, and CONST rules we have $\hat{\rho}_t(x) = \hat{\rho}_{in}(x)$. Hence, $\mathcal{G}_t(e,\ \hat{\rho}_{in}(x)) = \hat{\rho}_{in}(x) = \hat{\rho}_t(x)$.

Case 3. If $\hat{v} \sqcap \top_t \neq \bot$ and e is a function call $e_0\ e_1$, then:

- By the CALL$_{in}$ rule, we have $\hat{\rho}^{e_0}_{in}(x) = \hat{\rho}_{in}(x)$.
- By the induction hypothesis on e_0, we have $\hat{\rho}^{e_0}_t(x) = \mathcal{G}_t(e_0,\ \hat{\rho}^{e_0}_{in}(x))$ and $\hat{\rho}^{e_0}_f(x) = \mathcal{G}_f(e_0,\ \hat{\rho}^{e_0}_{in}(x))$. Since $\hat{v} \sqcap \top_t \neq \bot$ and thus $\hat{v}^{e_0} \neq \bot$, these simplify to $\hat{\rho}^{e_0}_t(x) = \hat{\rho}^{e_0}_{in}(x)$ and $\hat{\rho}^{e_0}_f(x) = \hat{\rho}^{e_0}_{in}(x)$.
- By the CALL$_{mid}$ rule, we have $\hat{\rho}^{e_1}_{in}(x) = \hat{\rho}^{e_0}_t(x) \sqcup \hat{\rho}^{e_0}_f(x)$.
- By the induction hypothesis on e_1, we have $\hat{\rho}^{e_1}_t(x) = \mathcal{G}_t(e_1,\ \hat{\rho}^{e_1}_{in}(x))$ and $\hat{\rho}^{e_1}_f(x) = \mathcal{G}_f(e_1,\ \hat{\rho}^{e_1}_{in}(x))$. Since $\hat{v} \sqcap \top_t \neq \bot$ and thus $\hat{v}^{e_1} \neq \bot$, these simplify to $\hat{\rho}^{e_1}_t(x) = \hat{\rho}^{e_1}_{in}(x)$ and $\hat{\rho}^{e_1}_f(x) = \hat{\rho}^{e_1}_{in}(x)$.
- By the CALL$_{out}$ rule, we have $\hat{\rho}^e_t(x) = ARG(\hat{\rho}^{e_1}_t \sqcup \hat{\rho}^{e_1}_f, e_1, \hat{v}^{e_1} \sqcap \hat{v}_t)(x)$. Since $\hat{v} \sqcap \top_t \neq \bot$, we have $\hat{v}^{e_1} \sqcap \hat{v}_t \neq \bot$. Since x is not in e, $\mathtt{x}$ is also not in e_1. Together these simplify the equation to $\hat{\rho}^e_t(x) = \left(\hat{\rho}^{e_1}_t \sqcup \hat{\rho}^{e_1}_f\right)(x)$.

Combining and simplifying these, we thus have $\mathcal{G}_t(e,\ \hat{\rho}_{in}(x)) = \hat{\rho}_{in}(x) = \hat{\rho}_t(x)$.

Case 4. If $\hat{v} \sqcap \top_t \neq \bot$ and e is a sequencing $e_0; e_1$, then:

- By the SEQ$_{in}$ rule we have $\hat{\rho}^{e_0}_{in}(x) = \hat{\rho}_{in}(x)$.

- By the induction hypothesis on e_0, we have $\hat{\rho}_t^{e_0}(x) = \mathcal{G}_t(e_0,\ \hat{\rho}_{in}^{e_0}(x))$ and $\hat{\rho}_f^{e_0}(x) = \mathcal{G}_f(e_0,\ \hat{\rho}_{in}^{e_0}(x))$. Since $\hat{v} \sqcap \top_t \neq \bot$ and thus $\hat{v}^{e_0} \neq \bot$, these simplify to $\hat{\rho}_t^{e_0}(x) = \hat{\rho}_{in}^{e_0}(x)$ and $\hat{\rho}_f^{e_0}(x) = \hat{\rho}_{in}^{e_0}(x)$.
- By the SEQ$_{mid}$ rule, we have $\hat{\rho}_{in}^{e_1}(x) = \hat{\rho}_t^{e_0}(x) \sqcup \hat{\rho}_f^{e_0}(x)$.
- By the induction hypothesis on e_1, we have $\hat{\rho}_t^{e_1}(x) = \mathcal{G}_t(e_1,\ \hat{\rho}_{in}^{e_1}(x))$. Since $\hat{v} \sqcap \top_t \neq \bot$ and thus $\hat{v}^{e_1} \neq \bot$, these simplify to $\hat{\rho}_t^{e_1}(x) = \hat{\rho}_{in}^{e_1}(x)$.
- By the SEQ$_{out}$ rule, we have $\hat{\rho}_t^{e}(x) = \hat{\rho}_t^{e_1}(x)$.

Combining and simplifying these, we thus have $\mathcal{G}_t(e,\ \hat{\rho}_{in}(x)) = \hat{\rho}_{in}(x) = \hat{\rho}_t(x)$.

Case 5. If $\hat{v} \sqcap \top_t \neq \bot$ and e is a conditional *if* $e_0\ e_1\ e_2$, then:

- By the IF$_{in}$ rule, we have $\hat{\rho}_{in}^{e_0}(x) = \hat{\rho}_{in}(x)$.
- By the induction hypothesis on e_0, we have $\hat{\rho}_t^{e_0}(x) = \mathcal{G}_t(e_0,\ \hat{\rho}_{in}^{e_0}(x))$ and $\hat{\rho}_f^{e_0}(x) = \mathcal{G}_f(e_0,\ \hat{\rho}_{in}^{e_0}(x))$. Since $\hat{v} \sqcap \top_t \neq \bot$ and thus $\hat{v}^{e_0} \neq \bot$, these simplify to $\hat{\rho}_t^{e_0}(x) = \hat{\rho}_{in}^{e_0}(x)$ and $\hat{\rho}_f^{e_0}(x) = \hat{\rho}_{in}^{e_0}(x)$.
- By the IF$_{mid}$ rule, we have $\hat{\rho}_{in}^{e_1}(x) = \hat{\rho}_t^{e_0}(x)$ and $\hat{\rho}_{in}^{e_2}(x) = \hat{\rho}_f^{e_0}(x)$.
- By the induction hypothesis on e_1 and e_2, we have $\hat{\rho}_t^{e_1}(x) = \mathcal{G}_t(e_1,\ \hat{\rho}_{in}^{e_1}(x))$, $\hat{\rho}_f^{e_1}(x) = \mathcal{G}_f(e_1,\ \hat{\rho}_{in}^{e_1}(x))$, $\hat{\rho}_t^{e_2}(x) = \mathcal{G}_t(e_2,\ \hat{\rho}_{in}^{e_2}(x))$, and $\hat{\rho}_f^{e_2}(x) = \mathcal{G}_f(e_2,\ \hat{\rho}_{in}^{e_2}(x))$. Since $\hat{v} \sqcap \top_t \neq \bot$, either $\hat{v}^{e_1} \neq \bot$, $\hat{v}^{e_2} \neq \bot$, or both.
  - If $\hat{v}^{e_1} \neq \bot$, then the first two equations simplify to $\hat{\rho}_t^{e_1}(x) = \hat{\rho}_{in}^{e_1}(x)$ and $\hat{\rho}_f^{e_1}(x) = \hat{\rho}_{in}^{e_1}(x)$. Otherwise they simplify to $\hat{\rho}_t^{e_1}(x) = \bot$ and $\hat{\rho}_f^{e_1}(x) = \bot$.
  - If $\hat{v}^{e_2} \neq \bot$, then the last two equations simplify to $\hat{\rho}_t^{e_2}(x) = \hat{\rho}_{in}^{e_2}(x)$ and $\hat{\rho}_f^{e_2}(x) = \hat{\rho}_{in}^{e_2}(x)$. Otherwise they simplify to $\hat{\rho}_t^{e_2}(x) = \bot$ and $\hat{\rho}_f^{e_2}(x) = \bot$.
- By the IF$_{out}$ rule, we have $\hat{\rho}_t^{e}(x) = (\hat{\rho}_t^{e_1} \sqcup \hat{\rho}_t^{e_2})\,(x)$.

Combining and simplifying these we thus have $\mathcal{G}_t(e,\ \hat{\rho}_{in}(x)) = \hat{\rho}_{in}(x) = \hat{\rho}_t(x)$ regardless of whether $\hat{v}^{e_1} \neq \bot$, $\hat{v}^{e_2} \neq \bot$, or both.

□

Going back to Example 4.1, this theorem allows the analysis to directly compute the abstract values of `x` at the end of e_1 and e_2 given the abstract values at the start of e_1 and e_2. The $\mathcal{G}_t(e,\ u)$ and $\mathcal{G}_f(e,\ u)$ functions act as guards and return the abstract value u if an only if $[\![e]\!]_{out}$ contains true or false values respectively.

This theorem deals only with the flow of abstract values from the entry of an expression to its exit. As seen in Example 4.1 and Example 4.2, however, we are also interested in how abstract values flow from an expression through its surrounding context. In the examples, abstract values flow from the outputs of `(pair? x)` and `(car x)` to the output of the surrounding `(if (pair? x)` e_1 e_2`)` and `(if` e_0 `(begin (car x)` e_1`)` e_2`)` respectively. To account for this, the analysis computes the flow across a context by means of the context-skipping function $\mathcal{V}_{C,e}$ defined in Figure 4.2. Given an expression, e, in a single-layer context, C, it computes the abstract value of a variable in the exit environments of $C[e]$ given the abstract value in the exit environments of e and the entry environment of $C[e]$. The following theorem states this formally. Again, the intuition is that the true and false information about a variable might join or swap but do not otherwise change.

Theorem 4.2. *Single-Layer Context Skipping (Exiting)*

If $\mathtt{x}$ *is a variable not mentioned in the single-layer context* C *then*

$$\left\langle \hat{\rho}_t^{C[e]}(x),\ \hat{\rho}_f^{C[e]}(x),\ \hat{\rho}_{in}(x) \right\rangle = \mathcal{V}_{C,e} \left\langle \hat{\rho}_t^{e}(x),\ \hat{\rho}_f^{e}(x),\ \hat{\rho}_{in}(x) \right\rangle$$

$$\textit{where } \left\langle \hat{v}^{e},\ \hat{\rho}_t^{e},\ \hat{\rho}_f^{e} \right\rangle = [\![e]\!]_{out}$$

$$\left\langle \hat{v}^{C[e]},\ \hat{\rho}_t^{C[e]},\ \hat{\rho}_f^{C[e]} \right\rangle = [\![C[e]]\!]_{out}$$

$$\langle \hat{r},\ \hat{\rho}_{in} \rangle = [\![C[e]]\!]_{in}.$$

$$\mathcal{V}_{C,e} \in \widehat{Val} \times \widehat{Val} \times \widehat{Val} \to \widehat{Val} \times \widehat{Val} \times \widehat{Val}$$
$$\mathcal{V}_{C,e}\,\langle \hat{v}_t,\, \hat{v}_f,\, \hat{v}_{in}\rangle = \left\langle \mathcal{G}_t(C[e]\,,\ \hat{v}'_t),\, \mathcal{G}_f(C[e]\,,\ \hat{v}'_f),\, \hat{v}'_{in} \right\rangle$$
$$\text{where } \left\langle \hat{v}'_t,\, \hat{v}'_f,\, \hat{v}'_{in} \right\rangle = \mathcal{V}'_C\,\langle \hat{v}_t,\, \hat{v}_f,\, \hat{v}_{in}\rangle$$

$$\begin{aligned}
\mathcal{V}'_{(\lambda x.\Box)}\,\langle \hat{v}_t,\, \hat{v}_f,\, \hat{v}_{in}\rangle &= \langle \hat{v}_{in},\, \hat{v}_{in},\, \hat{v}_{in}\rangle \\
\mathcal{V}'_{(\Box\ e_1)}\,\langle \hat{v}_t,\, \hat{v}_f,\, \hat{v}_{in}\rangle &= \langle \hat{v}_t \sqcup \hat{v}_f,\, \hat{v}_t \sqcup \hat{v}_f,\, \hat{v}_{in}\rangle \\
\mathcal{V}'_{(e_0\ \Box)}\,\langle \hat{v}_t,\, \hat{v}_f,\, \hat{v}_{in}\rangle &= \langle \hat{v}_t \sqcup \hat{v}_f,\, \hat{v}_t \sqcup \hat{v}_f,\, \hat{v}_{in}\rangle \\
\mathcal{V}'_{(\Box;e_1)}\,\langle \hat{v}_t,\, \hat{v}_f,\, \hat{v}_{in}\rangle &= \langle \hat{v}_t \sqcup \hat{v}_f,\, \hat{v}_t \sqcup \hat{v}_f,\, \hat{v}_{in}\rangle \\
\mathcal{V}'_{(e_0;\Box)}\,\langle \hat{v}_t,\, \hat{v}_f,\, \hat{v}_{in}\rangle &= \langle \hat{v}_t,\, \hat{v}_f,\, \hat{v}_{in}\rangle \\
\mathcal{V}'_{(\mathit{if}\ \Box\ e_1\ e_2)}\,\langle \hat{v}_t,\, \hat{v}_f,\, \hat{v}_{in}\rangle &= \left\langle \hat{v}'_t,\, \hat{v}'_f,\, \hat{v}_{in} \right\rangle \\
\text{where } \hat{v}'_t &= \mathcal{G}_t(e_1,\ \hat{v}_t) \sqcup \mathcal{G}_t(e_2,\ \hat{v}_f) \\
\hat{v}'_f &= \mathcal{G}_f(e_1,\ \hat{v}_t) \sqcup \mathcal{G}_f(e_2,\ \hat{v}_f) \\
\mathcal{V}'_{(\mathit{if}\ e_0\ \Box\ e_2)}\,\langle \hat{v}_t,\, \hat{v}_f,\, \hat{v}_{in}\rangle &= \left\langle \hat{v}'_t,\, \hat{v}'_f,\, \hat{v}_{in} \right\rangle \\
\text{where } \hat{v}'_t &= \hat{v}_t \sqcup \mathcal{G}_t(e_2,\ \hat{v}_{in}) \\
\hat{v}'_f &= \hat{v}_f \sqcup \mathcal{G}_f(e_2,\ \hat{v}_{in}) \\
\mathcal{V}'_{(\mathit{if}\ e_0\ e_1\ \Box)}\,\langle \hat{v}_t,\, \hat{v}_f,\, \hat{v}_{in}\rangle &= \left\langle \hat{v}'_t,\, \hat{v}'_f,\, \hat{v}_{in} \right\rangle \\
\text{where } \hat{v}'_t &= \hat{v}_t \sqcup \mathcal{G}_t(e_1,\ \hat{v}_{in}) \\
\hat{v}'_f &= \hat{v}_f \sqcup \mathcal{G}_f(e_1,\ \hat{v}_{in})
\end{aligned}$$

Figure 4.2.: Context skipping function

Proof. The proof proceeds by cases on C while using the constraint rules in Figure 3.5 and Theorem 4.1. Only the part for $\hat{\rho}_t^{C[e]}(x)$ is shown as the part for $\hat{\rho}_f^{C[e]}(x)$ is almost identical. For ease of discussion, let $\left\langle \hat{\rho}_t^{\mathcal{V}}(x),\, \hat{\rho}_f^{\mathcal{V}}(x),\, \hat{\rho}_{in}^{\mathcal{V}}(x) \right\rangle = \mathcal{V}_{C,e} \left\langle \hat{\rho}_t^{e}(x),\, \hat{\rho}_f^{e}(x),\, \hat{\rho}_{in}(x) \right\rangle$.

Case 1. If $\hat{v} \sqcap \top_t = \bot$, then $\hat{\rho}_t^{\mathcal{V}}(x) = \bot$. Thus, by the reduced abstract domain, we have $\hat{\rho}_t^{C[e]}(x) = \bot$. Hence, $\hat{\rho}_t^{C[e]}(x) = \hat{\rho}_t^{\mathcal{V}}(x)$.

Case 2. If $\hat{v} \sqcap \top_t \neq \bot$ and C is $(\lambda x'.\square)$, then by the LAMBDA rule, we have $\hat{\rho}_t^{C[e]}(x) = \hat{\rho}_{in}^{C[e]}(x) = \hat{\rho}_t^{\mathcal{V}}(x)$.

Case 3. If $\hat{v} \sqcap \top_t \neq \bot$ and C is $(\square\ e_1)$, then:

- By the definition of $\mathcal{V}_{C,e}$, we have $\hat{\rho}_t^{e}(x) \sqcup \hat{\rho}_f^{e}(x) = \hat{\rho}_t^{\mathcal{V}}(x)$.
- By the CALL$_{mid}$ rule, we have $\hat{\rho}_{in}^{e_1}(x) = \hat{\rho}_t^{e}(x) \sqcup \hat{\rho}_f^{e}(x)$.
- By Theorem 4.1, we have $\hat{\rho}_t^{e_1}(x) = \hat{\rho}_{in}^{e_1}(x)$ and $\hat{\rho}_f^{e_1}(x) = \hat{\rho}_{in}^{e_1}(x)$.
- By the CALL$_{out}$ rule, we have $\hat{\rho}_t^{C[e]}(x) = ARG(\hat{\rho}_t^{e_1} \sqcup \hat{\rho}_f^{e_1}, e_1, \hat{v}^{e_1} \sqcap \hat{v}_t)(x)$. Since $\hat{v} \sqcap \top_t \neq \bot$, we have $\hat{v}^{e_1} \sqcap \hat{v}_t \neq \bot$. Since x is not in $C[e]$, `x` is also not in e. Together these simplify the equation to $\hat{\rho}_t^{C[e]}(x) = \left(\hat{\rho}_t^{e_1} \sqcup \hat{\rho}_f^{e_1}\right)(x)$.

Combining and simplifying these, we thus have $\hat{\rho}_t^{C[e]}(x) = \hat{\rho}_t^{\mathcal{V}}(x)$.

Case 4. If $\hat{v} \sqcap \top_t \neq \bot$ and C is $(e_0\ \square)$, then:

- By the definition of $\mathcal{V}_{C,e}$, we have $\left(\hat{\rho}_t^{e} \sqcup \hat{\rho}_f^{e}\right)(x) = \hat{\rho}_t^{\mathcal{V}}(x)$.
- By the CALL$_{out}$ rule, we have $\hat{\rho}_t^{C[e]}(x) = ARG(\hat{\rho}_t^{e} \sqcup \hat{\rho}_f^{e}, e_1, \hat{v}^{e} \sqcap \hat{v}_t)(x)$. Since $\hat{v} \sqcap \top_t \neq \bot$, we have $\hat{v}^{e} \sqcap \hat{v}_t \neq \bot$. Since x is not in $C[e]$, `x` is also not in e. Together these simplify the equation to $\hat{\rho}_t^{C[e]}(x) = \left(\hat{\rho}_t^{e} \sqcup \hat{\rho}_f^{e}\right)(x)$.

Combining and simplifying these, we thus have $\hat{\rho}_t^{C[e]}(x) = \hat{\rho}_t^{\mathcal{V}}(x)$.

Case 5. If $\hat{v} \sqcap \top_t \neq \bot$ and C is $(\square; e_1)$, then:

- By the definition of $\mathcal{V}_{C,e}$, we have $\hat{\rho}_t^e(x) \sqcup \hat{\rho}_f^e(x) = \hat{\rho}_t^{\mathcal{V}}(x)$.
- By the SEQ_{mid} rule, we have $\hat{\rho}_{in}^{e_1}(x) = \hat{\rho}_t^e(x) \sqcup \hat{\rho}_f^e(x)$.
- By Theorem 4.1, we have $\hat{\rho}_t^{e_1}(x) = \mathcal{G}_t(e_1,\ \hat{\rho}_{in}^{e_1}(x))$. Since $\hat{v} \sqcap \top_t \neq \bot$ and thus $\hat{v}^{e_1} \neq \bot$, this simplifies to $\hat{\rho}_t^{e_1}(x) = \hat{\rho}_{in}^{e_1}(x)$.
- By the SEQ_{out} rule, we have $\hat{\rho}_t^{C[e]}(x) = \hat{\rho}_t^{e_1}(x)$.

Combining and simplifying these, we thus have $\hat{\rho}_t^{C[e]}(x) = \hat{\rho}_t^{\mathcal{V}}(x)$.

Case 6. If $\hat{v} \sqcap \top_t \neq \bot$ and C is $(e_0; \square)$, then:

- By the definition of $\mathcal{V}_{C,e}$, we have $\hat{\rho}_t^e(x) = \hat{\rho}_t^{\mathcal{V}}(x)$.
- By the SEQ_{out} rule, we have $\hat{\rho}_t^{C[e]}(x) = \hat{\rho}_t^e(x)$.

Combining and simplifying these, we thus have $\hat{\rho}_t^{C[e]}(x) = \hat{\rho}_t^{\mathcal{V}}(x)$.

Case 7. If $\hat{v} \sqcap \top_t \neq \bot$ and C is $(\mathit{if}\ \square\ e_1\ e_2)$, then:

- By the definition of $\mathcal{V}_{C,e}$, we have $\mathcal{G}_t(e_1,\ \hat{\rho}_t^e(x)) \sqcup \mathcal{G}_t(e_2,\ \hat{\rho}_f^e(x)) = \hat{\rho}_t^{\mathcal{V}}(x)$.
- By the IF_{mid} rule, we have $\hat{\rho}_{in}^{e_1}(x) = \hat{\rho}_t^e(x)$ and $\hat{\rho}_{in}^{e_2}(x) = \hat{\rho}_f^e(x)$.
- By Theorem 4.1, we have $\hat{\rho}_t^{e_1}(x) = \mathcal{G}_t(e_1,\ \hat{\rho}_{in}^{e_1}(x))$, $\hat{\rho}_f^{e_1}(x) = \mathcal{G}_f(e_1,\ \hat{\rho}_{in}^{e_1}(x))$, $\hat{\rho}_t^{e_2}(x) = \mathcal{G}_t(e_2,\ \hat{\rho}_{in}^{e_2}(x))$, and $\hat{\rho}_f^{e_2}(x) = \mathcal{G}_f(e_2,\ \hat{\rho}_{in}^{e_2}(x))$. Since $\hat{v} \sqcap \top_t \neq \bot$, either $\hat{v}^{e_1} \neq \bot$, $\hat{v}^{e_2} \neq \bot$, or both.
  - If $\hat{v}^{e_1} \neq \bot$, then the first two equations simplify to $\hat{\rho}_t^{e_1}(x) = \hat{\rho}_{in}^{e_1}(x)$ and $\hat{\rho}_f^{e_1}(x) = \hat{\rho}_{in}^{e_1}(x)$. Otherwise they simplify to $\hat{\rho}_t^{e_1}(x) = \bot$ and $\hat{\rho}_f^{e_1}(x) = \bot$.
  - If $\hat{v}^{e_2} \neq \bot$, then the last two equations simplify to $\hat{\rho}_t^{e_2}(x) = \hat{\rho}_{in}^{e_2}(x)$ and $\hat{\rho}_f^{e_2}(x) = \hat{\rho}_{in}^{e_2}(x)$. Otherwise they simplify to $\hat{\rho}_t^{e_2}(x) = \bot$ and $\hat{\rho}_f^{e_2}(x) = \bot$.
- By the IF_{out} rule we have $\hat{\rho}_t^{C[e]}(x) = (\hat{\rho}_t^{e_1} \sqcup \hat{\rho}_t^{e_2})(x)$.

Combining and simplifying these, we thus have $\hat{\rho}_t^{C[e]}(x) = \hat{\rho}_t^{\mathcal{V}}(x)$ regardless of whether $\hat{v}^{e_1} \neq \bot$, $\hat{v}^{e_2} \neq \bot$, or both..

Case 8. If $\hat{v} \sqcap \top_t \neq \bot$ and C is (*if* $e_0 \, \square \, e_2$), then:

- By the definition of $\mathcal{V}_{C,e}$, we have $\hat{\rho}_t^e(x) \sqcup \mathcal{G}_t(e_2,\ \hat{\rho}_{in}^{C[e]}(x)) = \hat{\rho}_t^{\mathcal{V}}(x)$.
- By the IF$_{in}$ rule, we have $\hat{\rho}_{in}^{e_0}(x) = \hat{\rho}_{in}^{C[e]}(x)$.
- By Theorem 4.1 on e_0, we have $\hat{\rho}_t^{e_0}(x) = \mathcal{G}_t(e_0,\ \hat{\rho}_{in}^{e_0}(x))$ and $\hat{\rho}_f^{e_0}(x) = \mathcal{G}_f(e_0,\ \hat{\rho}_{in}^{e_0}(x))$. Since $\hat{v} \sqcap \top_t \neq \bot$ and thus $\hat{v}^{e_0} \neq \bot$, these simplify to $\hat{\rho}_t^{e_0}(x) = \hat{\rho}_{in}^{e_0}(x)$ and $\hat{\rho}_f^{e_0}(x) = \hat{\rho}_{in}^{e_0}(x)$.
- By the IF$_{mid}$ rule, we have $\hat{\rho}_{in}^{e_2}(x) = \hat{\rho}_f^{e_0}(x)$.
- By Theorem 4.1 on e_2, we have $\hat{\rho}_t^{e_2}(x) = \mathcal{G}_t(e_2,\ \hat{\rho}_{in}^{e_2}(x))$ and $\hat{\rho}_f^{e_2}(x) = \mathcal{G}_f(e_2,\ \hat{\rho}_{in}^{e_2}(x))$. Since $\hat{v} \sqcap \top_t \neq \bot$, either $\hat{v}^{e_1} \neq \bot$, $\hat{v}^{e_2} \neq \bot$, or both.
  - If $\hat{v}^{e_2} \neq \bot$, then these two equations simplify to $\hat{\rho}_t^{e_2}(x) = \hat{\rho}_{in}^{e_2}(x)$ and $\hat{\rho}_f^{e_2}(x) = \hat{\rho}_{in}^{e_2}(x)$. Otherwise they simplify to $\hat{\rho}_t^{e_2}(x) = \bot$ and $\hat{\rho}_f^{e_2}(x) = \bot$.
- By the IF$_{out}$ rule, we have $\hat{\rho}_t^e(x) = (\hat{\rho}_t^{e_1} \sqcup \hat{\rho}_t^{e_2})\,(x)$.

Combining and simplifying these, we thus have $\hat{\rho}_t^{C[e]}(x) = \hat{\rho}_t^{\mathcal{V}}(x)$ regardless of whether $\hat{v}^{e_2} \neq \bot$.

Case 9. If $\hat{v} \sqcap \top_t \neq \bot$ and C is (*if* $e_0 \, e_1 \, \square$), then the same reasoning as for (*if* $e_0 \, \square \, e_2$) applies except that e_1 and e_2 swap roles.

□

For example, consider `(if (pair? x)` e_1 e_2`)`. The context of `(pair? x)` is (*if* $\square \, e_1 \, e_2$). Thus, if e_1 returns only true values and e_2 returns only false values, the skipping function, $\mathcal{V}_{C,e}$, does not change the values in the environment as they flow from `(pair? x)`. On the other hand, if e_1 returns only false values and e_2 returns only true values, the skipping

function swaps the values of the true and false environments. Other possibilities arise depending on the values returned by e_1 and e_2.

While Theorem 4.2 handles only single layer contexts, the following theorem generalizes this to multilayer contexts.[1]

Theorem 4.3. *Multilayer Context Skipping (Exiting)*

If $\mathbf{x}$ is a variable not mentioned in a single-layer or multilayer context C then the equation from Theorem 4.2 holds where $\mathcal{V}_{C,e}$ on a composite context is

$$\mathcal{V}_{C_2C_1,e} = \mathcal{V}_{C_2,C_1[e]} \circ \mathcal{V}_{C_1,e}$$

Proof. By induction on C and Theorem 4.2. □

Note that even for multilayer contexts the universe of possible $\mathcal{V}_{C,e}$ is finite and small. Any particular $\mathcal{V}_{C,e}$ can be represented in a canonical form of constant size in the following theorem.

Theorem 4.4. *Canonical Skipping Functions*

There exist $T, F \subseteq \{\hat{v}_t, \hat{v}_f, \hat{v}_{in}\}$ for any C, e, $[\![e]\!]_{out}$ and $[\![C[e]]\!]_{out}$ such that

$$\mathcal{V}_{C,e}\,\langle \hat{v}_t,\, \hat{v}_f,\, \hat{v}_{in}\rangle = \left\langle \bigsqcup T,\, \bigsqcup F,\, \hat{v}_{in} \right\rangle$$

Proof. By induction on C and unfolding $\mathcal{V}_{C,e}$. □

Intuitively, there are only so many ways to join and swap the true and false values of a variable. Diagrammatically, these canonical forms are all sub-graphs of the graph in Figure 4.3 that omit zero or more of the dashed edges that lead to the two join ($\sqcup$) nodes. Functions of this form have compact, constant-size representations, are closed

[1] This theorem is the reason why the tuple returned by $\mathcal{V}_{C,e}$ has a third component even though it is unused in Theorem 4.2.

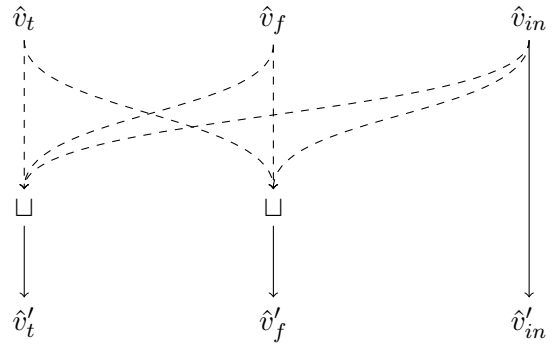


Figure 4.3.: Graph form of canonical skipping functions

under composition, and form a finite height lattice that they monotonically climb when $[\![e]\!]_{out}$ and $[\![C[e]]\!]_{out}$ climb the value lattice.

When composing $\mathcal{V}_{C,e}$ we always reduce the composition to this canonical form. Thus, all $\mathcal{V}_{C,e}$ can be applied to a given value in constant time even if the $\mathcal{V}_{C,e}$ is from the composition of many $\mathcal{V}_{C,e}$. The analysis takes advantage of this to flow abstract values quickly across multilayer contexts. Since $\mathcal{V}_{C,e}$ is the same for all variables not in C, the analysis can compute $\mathcal{V}_{C,e}$ once and use it for all variables not in C. Section 4.4 shows how to compute and update these compositions efficiently.

This skipping function is the key insight to the implementing a fast version of the analysis. The analysis still must choose which contexts to skip and how to compute $\mathcal{V}_{C,e}$ efficiently, but those aspects of the algorithm exists only so that the algorithm can use skipping functions to flow information through the program more efficiently.

4.2. Context skipping and reachability

Theorem 4.2 and Theorem 4.3 deal with environments exiting a context, i.e., $\hat{\rho}_t$ and $\hat{\rho}_f$. Theorem 4.5 and Theorem 4.6 are their duals and deal with the environment entering a

context. They are simpler and do not need to define a skipping function as a simpler reachability check suffices.

Theorem 4.5. *Single-Layer Context Skipping (Entering)*

If $\boldsymbol{x}$ *is a variable not mentioned in the single-layer context* C *then*

$$\hat{\rho}^e_{in}(x) = \begin{cases} \hat{\rho}^{C[e]}_{in}(x) & \text{if } \hat{r}^e = \top \\ \bot & \text{if } \hat{r}^e = \bot \end{cases}$$

$$\text{where } \langle \hat{r}^e,\, \hat{\rho}^e_{in} \rangle = [\![e]\!]_{in}$$

$$\left\langle \hat{r}^{C[e]},\, \hat{\rho}^{C[e]}_{in} \right\rangle = [\![C[e]]\!]_{in}$$

Proof. The proof proceeds by cases on C while using the constraint rules in Figure 3.5 and Theorem 4.1.

Case 1. If $\hat{r} = \bot$, then by the reduced abstract domain, we have $\hat{\rho}^e_{in}(x) = \bot$.

Case 2. If $\hat{r}_e = \top$ and C is $(\lambda x'.\square)$, then by the LAMBDA rule, we have $\hat{\rho}^e_{in}(x) = \hat{\rho}^{C[e]}_{in}(x)$.

Case 3. If $\hat{r}_e = \top$ and C is $(\square\ e_1)$, then by the CALL$_{in}$ rule, we have $\hat{\rho}^e_{in}(x) = \hat{\rho}^{C[e]}_{in}(x)$.

Case 4. If $\hat{r}_e = \top$ and C is $(e_0\ \square)$, then:

- By the CALL$_{in}$ rule, we have $\hat{\rho}^{e_0}_{in}(x) = \hat{\rho}^{C[e]}_{in}(x)$.
- By Theorem 4.1 on e_0, we have $\hat{\rho}^{e_0}_t(x) = \mathcal{G}_t(e_0,\ \hat{\rho}^{e_0}_{in}(x))$ and $\hat{\rho}^{e_0}_f(x) = \mathcal{G}_f(e_0,\ \hat{\rho}^{e_0}_{in}(x))$. Since $\hat{r}_e = \top$ and thus $\hat{r}_{e_0} = \top$, these simplify to $\hat{\rho}^{e_0}_t(x) = \hat{\rho}^{e_0}_{in}(x)$ and $\hat{\rho}^{e_0}_f(x) = \hat{\rho}^{e_0}_{in}(x)$.
- By the CALL$_{mid}$ rule, we have $\hat{\rho}^e_{in}(x) = \hat{\rho}^{e_0}_t(x) \sqcup \hat{\rho}^{e_0}_f(x)$.

Combining and simplifying these, we thus have $\hat{\rho}^e_{in}(x) = \hat{\rho}^{C[e]}_{in}(x)$.

Case 5. If $\hat{r}_e = \top$ and C is $(\square; e_1)$, then by the SEQ$_{in}$ rule, we have $\hat{\rho}^e_{in}(x) = \hat{\rho}^{C[e]}_{in}(x)$.

Case 6. If $\hat{r}_e = \top$ and C is $(e_0; \Box)$, then:

- By the SEQ_{in} rule, we have $\hat{\rho}_{in}^{e_0}(x) = \hat{\rho}_{in}^{C[e]}(x)$.
- By Theorem 4.1 on e_0, we have $\hat{\rho}_t^{e_0}(x) = \mathcal{G}_t(e_0,\ \hat{\rho}_{in}^{e_0}(x))$ and $\hat{\rho}_f^{e_0}(x) = \mathcal{G}_f(e_0,\ \hat{\rho}_{in}^{e_0}(x))$. Since $\hat{r}_e = \top$ and thus $\hat{r}_{e_0} = \top$, these simplify to $\hat{\rho}_t^{e_0}(x) = \hat{\rho}_{in}^{e_0}(x)$ and $\hat{\rho}_f^{e_0}(x) = \hat{\rho}_{in}^{e_0}(x)$.
- By the SEQ_{mid} rule, we have $\hat{\rho}_{in}^{e}(x) = \hat{\rho}_t^{e_0}(x) \sqcup \hat{\rho}_f^{e_0}(x)$.

Combining and simplifying these we thus have $\hat{\rho}_{in}^{e}(x) = \hat{\rho}_{in}^{C[e]}(x)$.

Case 7. If $\hat{r}_e = \top$ and C is $(\mathit{if}\ \Box\ e_1\ e_2)$, then by the IF_{in} rule, we have $\hat{\rho}_{in}^{e}(x) = \hat{\rho}_{in}^{C[e]}(x)$.

Case 8. If $\hat{r}_e = \top$ and C is $(\mathit{if}\ e_0\ \Box\ e_2)$ or $(\mathit{if}\ e_0\ e_1\ \Box)$, then:

- By the IF_{in} rule, we have $\hat{\rho}_{in}^{e_0}(x) = \hat{\rho}_{in}^{C[e]}(x)$.
- By Theorem 4.1 on e_0, we have $\hat{\rho}_t^{e_0}(x) = \mathcal{G}_t(e_0,\ \hat{\rho}_{in}^{e_0}(x))$ and $\hat{\rho}_f^{e_0}(x) = \mathcal{G}_f(e_0,\ \hat{\rho}_{in}^{e_0}(x))$. Since $\hat{r}_e = \top$ and thus $\hat{r}_{e_0} = \top$, these simplify to $\hat{\rho}_t^{e_0}(x) = \hat{\rho}_{in}^{e_0}(x)$ and $\hat{\rho}_f^{e_0}(x) = \hat{\rho}_{in}^{e_0}(x)$.
- By the IF_{mid} rule, we have $\hat{\rho}_{in}^{e_1}(x) = \hat{\rho}_t^{e_0}(x)$ if C is $(\mathit{if}\ e_0\ \Box\ e_2)$ and $\hat{\rho}_{in}^{e_2}(x) = \hat{\rho}_f^{e_0}(x)$ if C is $(\mathit{if}\ e_0\ e_1\ \Box)$.

Combining and simplifying these we thus have $\hat{\rho}_{in}^{e}(x) = \hat{\rho}_{in}^{C[e]}(x)$. □

Theorem 4.6. *Multilayer Context Skipping (Entering)*

If $\mathtt{x}$ is a variable not mentioned in a single-layer or multilayer context C then the equation from Theorem 4.5 holds.

Proof. By induction on C and Theorem 4.5. □

4.3. Selecting context skips

The analysis now has two ways to flow abstract values through a program to be analyzed. The first is via the constraint rules in Figure 3.5. The second is via the skipping function,

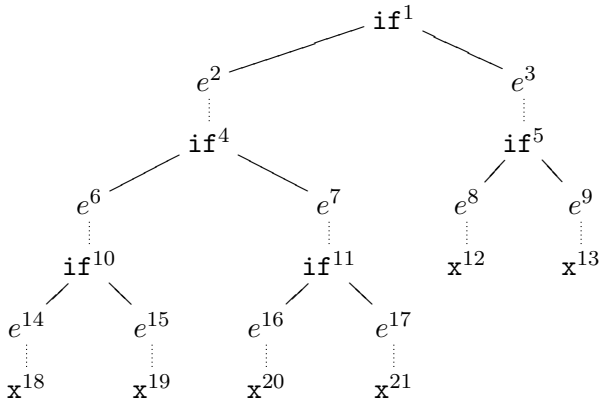


Figure 4.4.: Example AST for skipping context selection

$\mathcal{V}_{C,e}$. For each variable, the analysis uses a combination of these methods that ensures the analysis takes only linear-log time while maintaining semantic equivalence with the original algorithm.

This is done by selecting the longest contexts to be skipped for which Theorem 4.3 is valid for a given variable. The length of a context is measured by the number of layers in the context. The analysis falls back to the constraint rules in Figure 3.5 when the preconditions of Theorem 4.3 do not hold.

A different set of skips is selected for each variable, and the analysis selects longest skips for a particular variable, `x`, by starting with each reference to it and finding the longest context, C, of it that does not contain `x`. C is then one of the contexts to be skipped.

Since C is the longest context of e not containing `x`, the parent of $C[e]$, p, contains references to `x` other than the ones in e. Thus, the analysis cannot use Theorem 4.3 to skip past p, and at p it falls back to the constraint rules from Figure 3.5. The analysis repeats the process by finding the longest context of p that does not contain `x` and chooses that context as one to be skipped. This repeats until the analysis has all the skips needed to flow `x` through the entire program.

As an example, consider the abstract syntax tree in Figure 4.4 and the longest skips for `x`. The dotted edges in the diagram represent multiple layers of the abstract syntax tree that are omitted and which do not contain references to `x`. The two children of each `if` are the consequent and the alternative. The test part of `if` if omitted for simplicity.

To select skips, all references to `x` are examined. In this case they are expressions 12, 13, 18, 19, 20 and 21. For each such expression, the longest context not containing `x` is selected. For expression 12, this is the context going from expression 12 to just past expression 8. For expression 13, this is the context going just past expression 9, and so on. The parents of these contexts are places where the constraint rules are used instead of the context skipping function. For example, because of the reference to `x` in expression 9, Theorem 4.3 is not valid for moving type information for `x` from expression 8 to its parent, expression 5. Thus, the context skipping function cannot be used there.

The process repeats with the parents of each of the skips. For example, expression 5 is the parent of the contexts ending at expressions 8 and 9 so the algorithm selects the longest context of expression 5 that does not contain `x`. Likewise for expressions 10 and 11.

In the end the only places where the algorithm uses the constraint rules are expressions 1, 4, 5, 10, and 11. Everywhere else it uses context skipping functions. The entire scope of `x` is thus tessellated by the skipping contexts and the points where the analysis falls back to the constraint rules.

This part of the algorithm is linear because the selected context skips form a tree structure. The expressions at which the analysis uses the constraint rules are the nodes of the tree. The contexts being skipped are the edges of the tree. The references to variables are the leaves. Since the number of edges and the number of nodes in a tree are both linearly bounded by the number of leaves, the number of skips and the number of uses of the constraint rules for a particular variable are both linearly bounded by the number of

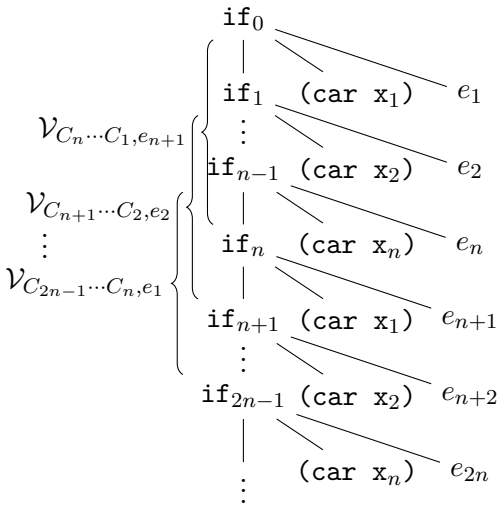


Figure 4.5.: Example of quadratic $\mathcal{V}_{C,e}$ calculation

references to that variable. Summing over all variables this is thus linear in the size of the program.

Finding the longest context not containing a particular variable is the most computationally complex part of this process. It is implemented in terms of a lowest common ancestor algorithm [Aho et al., 1973, Alstrup et al., 2004] that takes linear time for construction and constant time for each query. Finding the longest skips amounts to finding the lowest common ancestor of an expression and the immediately preceding and following references to the variable being considered.

4.4. Caching context skips

Theorem 4.3 allows the analysis to skip over a context and move information about variables quickly across multiple layers. Once the skipping function, $\mathcal{V}_{C,e}$, is computed and reduced to the canonical form in Theorem 4.4, it takes only constant time to move information across C for any variable not referenced in C.

We must be careful that the total time to construct all the skipping functions does not exceed our linear-log time bound. For example, consider the abstract syntax tree in Figure 4.5, where a different $\mathcal{V}_{C,e}$ is needed for each of the n variables, and each context is n layers deep. Computing the $\mathcal{V}_{C,e}$ for each variable separately takes $O(n^2)$ time.

To ensure a linear-log time bound, the analysis keeps a cache of $\mathcal{V}_{C,e}$ for selected C such that

- only linear-log many $\mathcal{V}_{C,e}$ are stored in the cache,
- for any C, a $\mathcal{V}_{C,e}$ can be computed from the composition of only logarithmically many $\mathcal{V}_{C,e}$ from the cache, and
- when more information is learned about an expression, only logarithmically many $\mathcal{V}_{C,e}$ in the cache need to be updated, and each $\mathcal{V}_{C,e}$ takes only constant time to update.

Figure 4.6 shows an example of the cached values for one path down a program's abstract syntax tree. Together all of these cached $\mathcal{V}_{C,e}$ form a tree structure. The same structure occurs on all other paths down the program tree. Each $\mathcal{V}_{C,e}$ shared between different paths is stored only once in the cache.

The cache can be thought of as starting with the $\mathcal{V}_{C,e}$ for single-layer contexts. That is, it stores the skipping information necessary to flow any variable by a single step from the exit environment of one expression to the enclosing expression's exit environment. If the cache stores only these, then when the abstract value of an expression changes, it takes only constant time to update.

Next, the single-layer $\mathcal{V}_{C,e}$ are paired together. Each single layer C that goes from depth $2k$ to depth $2k+1$ is paired with each of its single-layer, child contexts which go from depth $2k+1$ to $2k+2$. The $\mathcal{V}_{C,e}$ for each of these pairings is included in the cache. These double-layer $\mathcal{V}_{C,e}$ are then also paired together. Each double-layer C that goes from depth

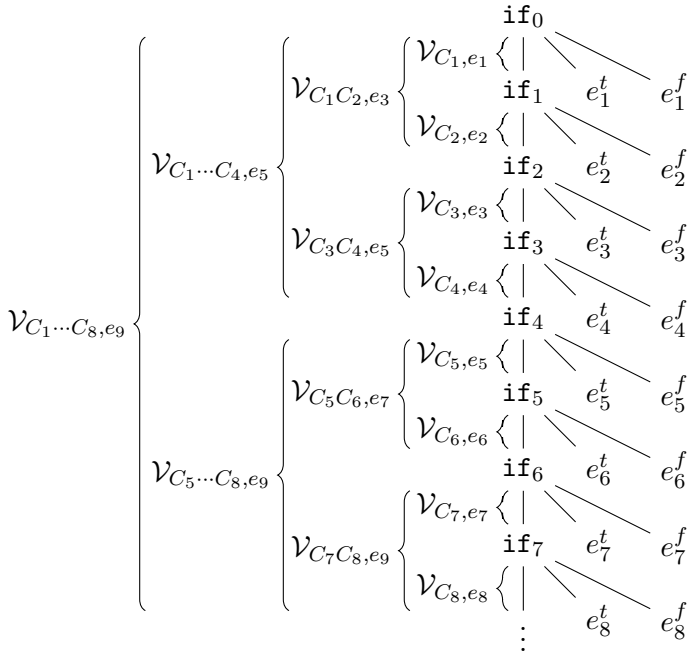


Figure 4.6.: Layered structure of the $\mathcal{V}_{C,e}$ cache

$4k$ to depth $4k + 2$ is paired with each of its double-layer, child contexts which go from depth $4k + 2$ to $4k + 4$. The $\mathcal{V}_{C,e}$ for each of these pairings is also included in the cache. This process continues iteratively, pairing each 2^i-layer context that goes from depth $2^{i+1}k$ to depth $2^{i+1}k + 2^i$ with each of its 2^i-layer child contexts which go from depth $2^{i+1}k + 2^i$ to $2^{i+1}k + 2^{i+1}$.

This selection of cached $\mathcal{V}_{C,e}$ has the three important properties that ensure our linear-logarithmic bound. First, only $O(n \log n)$ skipping functions are cached, since only logarithmically many $\mathcal{V}_{C,e}$ are cached for any particular e. Second, any $\mathcal{V}_{C,e}$ that is not cached can be computed from the composition of logarithmically many cached $\mathcal{V}_{C,e}$. Third, when the $\mathcal{V}_{C,e}$ for a single-layer context is updated, the double-layer $\mathcal{V}_{C,e}$ composed from it are also updated. If a double-layer $\mathcal{V}_{C,e}$ changes, the quadruple-layer $\mathcal{V}_{C,e}$ composed from it are updated, and so on. Thus, when abstract value information is learned about an expression, at most logarithmically many $\mathcal{V}_{C,e}$ in the cache are updated. Each update takes constant time since each multilayer $\mathcal{V}_{C,e}$ in the cache is composed of exactly two $\mathcal{V}_{C,e}$.

This caching strategy can be generalized by considering the path from each expression to the root. Storing this path as a perfectly balanced variation of a skip list [Pugh, 1990] is equivalent to the caching strategy just described. By using a variation of Myers applicative random access stacks [Myers, 1984], however, the number of cached values and the total time spent updating the cache both become linear in the size of the program. For an arbitrary C, computing $\mathcal{V}_{C,e}$ may still require logarithmically many cached values, so this does not improve the overall asymptotic bounds, but it improves the constants involved. This is the representation used by the implementation described in Chapter 6.

Though a single perfectly balanced skip list takes space linear in the length of the skip list, it degenerates when multiple skip lists share the same tail. Figure 4.7 shows such a perfectly balanced skip list. Each node, i, occupies $P(i)$ pointers worth of space where $P(i)$ is the largest integer such that $2^{P(i)-1}$ evenly divides i. Nodes of size greater than or

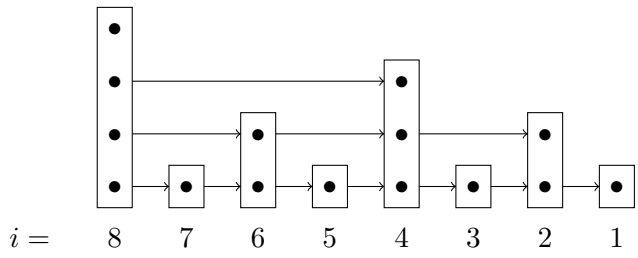


Figure 4.7.: A perfectly balanced skip list

equal to k happen once every 2^{k-1} nodes. That is, size two or larger nodes occur at every second node, size three or larger nodes occur at every fourth node, and so on. Thus, a single perfectly balanced list of length $n = 2^j$ has a size bounded by $\sum_{0 \leq k \leq j} \frac{2^j}{2^k} = \sum_{0 \leq k \leq j} 2^{j-k} = \sum_{0 \leq l \leq j} 2^l = 2^{j+1} - 1 = 2n - 1$.

This linear result does not hold, however, when multiple lists share a common tail. Figure 4.8 shows an example. It consists of a common tail that is $n = 2^i - 1$ nodes long and m different nodes attached at the head of the common tail. Each head contains $\lg(n+1)$ pointers. Thus, the tail has size $2n - 1$ and the heads together have size $2m \lg(n+1)$ for a total size of $2n + 2m \lg(n+1) - 1$. This is asymptotically greater than linear in the number of nodes. The size is still linear-logarithmically bounded by the number of nodes since each node has no more than a logarithmic number of nodes following it.

Unlike a skip list, a Myers applicative random access stack [Myers, 1984] takes linear space even when multiple stacks share the same tail. Instead of containing a variable number of pointers in each node, a Myers stack always contains exactly two pointers in each node. Thus, even when a set of stacks share tails, the total size is $2n$ where n is the total number of nodes.

Remarkably, the structure of a Myers stack still allows the analysis to compute the skipping function across any range in only $O(\log n)$ function compositions. Figure 4.9 shows the structure of a Myers stack. One can think of it as a flattening of a standard

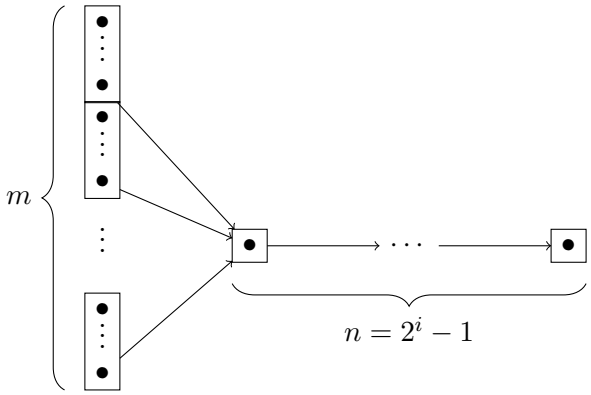


Figure 4.8.: Skip lists sharing a tail

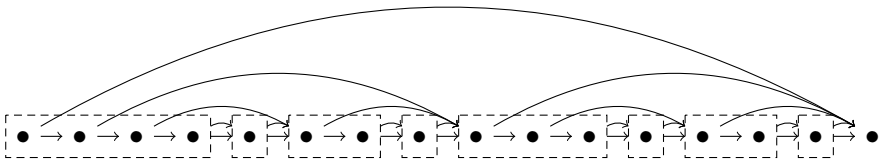


Figure 4.9.: A Myers stack

perfectly balanced skip list. Where a skip list has one node at index i with $P(i)$ pointers, a Myers stack has a cluster of $P(i)$ nodes which each have two pointers. One pointer is the *next pointer* and points to the next node in the stack. In Figure 4.9, the *next pointers* are all shown with straight, horizontal arrows. The other pointer is the *jump pointer* and points to the same node that the corresponding node in the skip list would. In Figure 4.9, the *jump pointers* are all arcing arrows.

To traverse from any node in a Myers stack to any following node, follow the jump pointer if it does not point past the destination and follow the next pointer otherwise. This requires traversing only $O(\log n)$ pointers where n is the number of nodes after the starting node. The proof of this follows similar reasoning to the case of perfectly balanced skip lists. For details of the proof see Myers [1984].

To use a myers stack as a cache for the contextual skipping functions, the analysis stores a skipping function along with each pointer. The next pointers store the skipping function for single-layer contexts. The jump pointers store the composition of the skipping functions in the range being jumped over. Since, as shown in Figure 4.9, each jump pointer jumps over a next pointer followed by two other jump pointers, the skipping function associated with a particular jump pointer is computed as the composition of those three functions. Thus, when the cache is represented as a flow graph there are a linear number of nodes because there are a linear number of pointers and associated skipping functions. Also there are a linear number of flow-graph edges because each node for a jump pointer has exactly three other nodes from which it is computed.

To summarize, a group of Myers stacks takes only $O(n)$ space, where n is the total number of nodes, even when tails are shared. They also allow traversal between any two points in only $O(\log n)$ time. When a Myers stack is used as a cache for skipping functions, each pointer is annotated with the appropriate skipping function. Thus, the cache takes linear space in the number of nodes, and thus is linear in the size of the abstract syntax tree

upon which the cache is overlaid. The next pointers are annotated with skipping functions directly computed as in Figure 4.2 for the single layer contexts spanned by the pointers. The jump pointers are computed from the composition of the functions associated with the three contained pointers. When represented in a flow graph, the graph nodes for these functions are thus linear in number, have a constant number of incoming edges per node, and can be computed in constant time. Thus, the cache takes time and space linear in the size of the program being analyzed.

While the skipping functions in the cache take only linear time and space, the skipping functions computed from the cache each require the composition of up to logarithmically many skipping functions. Since, as shown in Section 4.3, there are at most a linear number of non-cached skipping functions thus computed, these function contribute no more than $O(n \log n)$ time and space to the total time and space for the analysis. This is the only part of the analysis that takes more than linear time or space. Future asymptotic improvements here would directly improve the asymptotic performance of the whole analysis.

4.5. Algorithm Summary

Putting all these pieces together, the optimized algorithm works as follows. First, as described in Section 4.4, the cache of skipping functions is constructed as a flow graph. This creates linearly many nodes in linear time. Next, for each variable, context skips are selected as described in Section 4.3 and flow-graph nodes are constructed that take logarithmically many $\mathcal{V}_{C,e}$ from the cache and build a $\mathcal{V}_{C,e}$ for the skipped context. In total there are linearly many context skips and each one involves composing logarithmically many skipping functions. Each composition requires one flow-graph node, so this process creates $O(n \log n)$ nodes in $O(n \log n)$ time. Finally, for each non-skipping point where a variable is referenced or the constraint rules are used for a particular variable, a flow-graph

node is constructed that computes the type of the variable at that point in terms of the non-skipping points that flow to the point and the $\mathcal{V}_{C,e}$ that skips from them to the non-skipping point. A similar process is used for entering rather than exiting a context. Since in total there are linearly many skipping points, this creates linearly many nodes. Overall, this entire process takes linear-log time to construct the flow graph, and it produces a flow graph with a linear-log number of nodes. The values flowing over the edges of the graph all monotonically increase over constant-height lattices, and nodes recompute in terms of their inputs in constant time. Thus, the flow-graph for the optimized analysis converges in linear-log time.

As an example of this, consider the following expression where each C_i is a context that does not contain `x`. Assume, each e_i is an expression that does not contain `x`.

```
(if C1[(if C2[(if (pair? x) e1 e2)]                    (4.3)
           C3[(car x)]
           e3)]
  e4
  C4[(cdr x)])
```

The abstract syntax tree for this expression is as shown in Figure 4.10. Each C_i and ellipsis is an elided context. We use the convention mentioned before and say that we compute a value when we mean that a flow-graph node is constructed that computes the value.

The first step constructs flow-graph nodes to compute skipping functions over this tree as described in Section 4.4. For example, the path from `(pair? x)` to the root is overlaid by a Myers stack that contains flow-graph nodes for the skipping functions along the path. Likewise, Myers stacks are constructed for e_1, e_2 and all the other expressions. These Myers stacks share nodes when there are common tails. For example, `(pair? x)`, e_1 and e_2 all share a common tail, and `(car x)` also shares a tail but at a different point.

The second step selects which context skips to use for each variable, as described in Section 4.3. Assuming `x` is not in any C_i or e_i, the skips selected are those marked in Figure 4.11. The actual expression does distinguish each C_i from any other expression, so

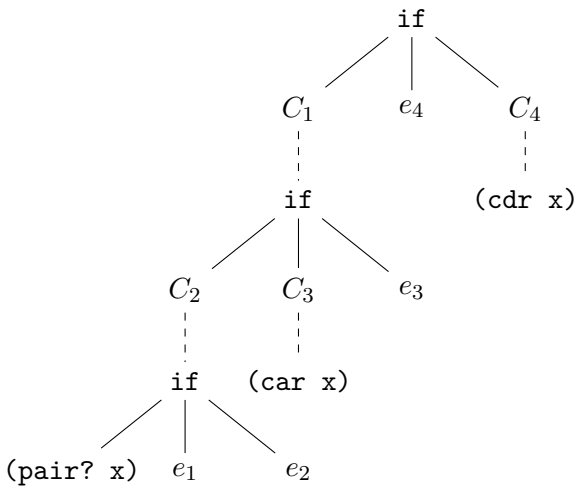


Figure 4.10.: Example AST

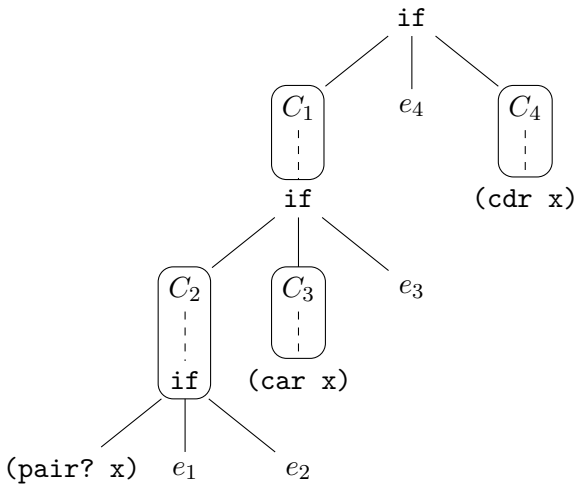


Figure 4.11.: Example AST with skips

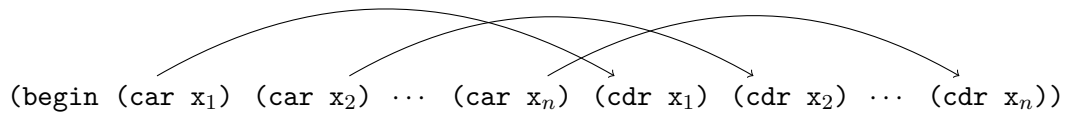


Figure 4.12.: Worst case for Myers stacks

the skips are computed in terms of the lowest common ancestors of each reference to `x`. The skips thus selected are the longest contexts not containing `x`.

Finally, to flow the value of `x` through the expression, a flow-graph node is constructed that takes the types of `x` for both the true and false expression cases coming out of `(pair? x)` and passes them to the skipping function computed for the context C_2[`(if` $\square$ e_1 e_2`)`]. Then, at the middle `if`, the standard constraint rules are used to flow the types for `x` when C_2[`(if (pair? x)` e_1 e_2`)`] is true back down to `(car x)`. Leaving `(car x)`, we have a flow-graph node that applies the skipping function for C_3 to the types for `x` leaving `(car x)`. Then the constraint rules for `if` are again used to flow the types of `x` up and out of the middle `if`. Then again the skipping function for C_1 is used to flow `x` up to the outer `if`. Finally, the constraint rules are used to flow the types of `x` across the `if` and down to the `(cdr x)`.

An example of the worst-case, $O(n \log n)$ behavior consider Figure 4.12. It represents a branch of an expression's abstract syntax tree with $2n$ nodes. At each node, one of n different variables is referenced and each variable is referenced at exactly two nodes that are n steps apart. When the skipping cache is overlaid on this, it takes only $O(n)$ space. To compute the skipping function between two references to a particular $\mathtt{x}_i$ requires composing $O(\log n)$ skipping functions from the cache. Since the expression has $O(n)$ variables which each use $O(\log n)$ compositions, we thus have the $O(n \log n)$ worst case.

5. Example

Because it is difficult to get a good picture of how the algorithm works just from the formal rules in Chapter 3 and the optimization to the algorithm described in Chapter 4, this chapter walks through an example and shows the flow graph produced by the optimized algorithm. The example is contrived to demonstrate several facets of the algorithm.

5.1. Code

This chapter uses the following code an example. The code contains examples of user-defined unconditional functions (`show-car`) and conditional predicates (`not-pair?`). Predicates and operators are deeply nested in both the test and consequent of conditionals. Variables appear not only in both branches of tests but also only in one branch of tests.

The flow graph resulting from this example is fairly large and intricate, so it is presented in stages and parts are simplified when they are not relevant to the optimized algorithm in Chapter 4.

5.2. Abstract syntax tree

Figure 5.2 shows the abstract syntax tree corresponding to the code in Figure 5.1.

The `let` consists of variable bindings and a body. The left-hand sides are connected to right-hand sides via dashed edges. The body is the rightmost child of the `let`.

```
(let ([x (read)]
      [y (read)]
      [z (read)]
      [not-pair? (lambda (a) (if (pair? a) #f #t))]
      [show-car (lambda (b) (display (car b)))])
  (if (if (if (if (if (if (not-pair? x) #f #t)
                      #f #t)
                  #f #t)
              #f #t)
          (if (read)
              (if (read)
                  (if (read)
                      (begin (cadr x) (not-pair? y))
                      #f)
                  #f)
              #f)
          (begin (read) (read) (read) (show-car y) (not-pair? z)))
      (begin (car x) (car y) (car z))
      (begin (car x) (car y) (car z))))
```

Figure 5.1.: Example expression

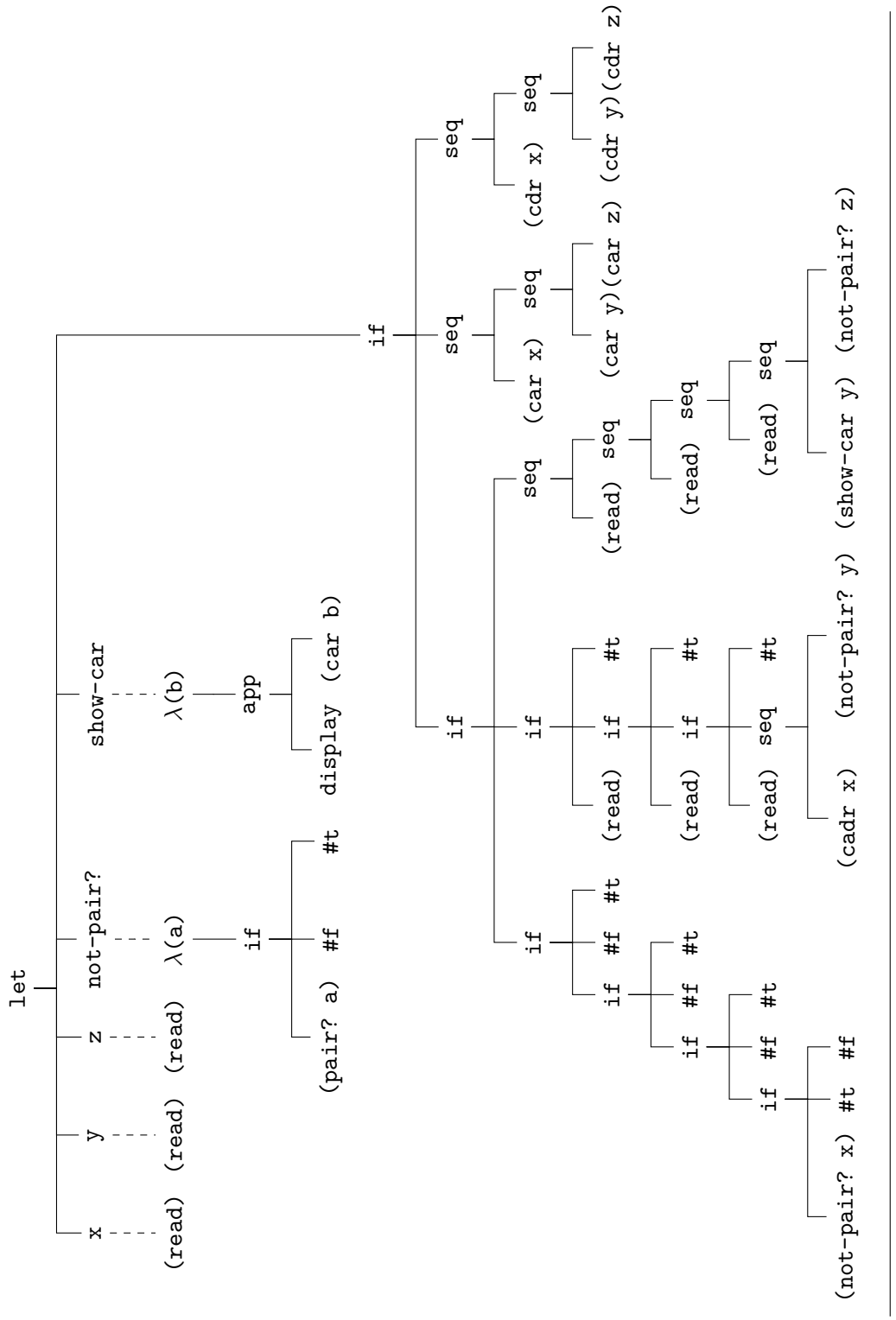


Figure 5.2.: Abstract syntax tree

Constants such as `#f` or `#t` are represented by themselves.

Simple function calls such as `(read)` or `(pair? a)` do not have their internall structure shown in this diagram as it does not affect the optimizations from Chapter 4. The only complex function call, `(display (car b))` is represented by `app` node with children for the function and argument positions of the call.

The `lambda` functions are represented by a λ`(x)` node where `x` is the variable bound by the function. The body of the function is the child of the λ`(x)` node.

The `if` conditional is represented by an `if` node that has three children. From left to right, the children are the test, consequent and alternative expressions of the conditional.

Sequencing is represented by a `seq` node with two children. The left child is the first expression to be evaluated. The right child is the second expression to be evaluated. Scheme's `begin` form is represented by nested `seq` nodes.

5.3. Value graphs

The flow graph for Figure 5.1 can be thought of as one portion for the reachability and result values and another portion for variables. Figure 5.3 shows the first portion overladed on top of the abstract syntax tree.

In Figure 5.3, each circle represents a flow-graph node, and each arrow represents a flow-graph edge. An edge with a single-headed arrowhead is for reachability information. An edge with a double-headed arrowhead is for expression result values. Finally, each node contains a letter indicating its role in the analysis. The meanings of these letters are as follows.

r: The reachability of an expression. This is used both for the root expression and the body of each function. The root expression is always reachable. A function body is reachable if and only if that function is called at some function call site.

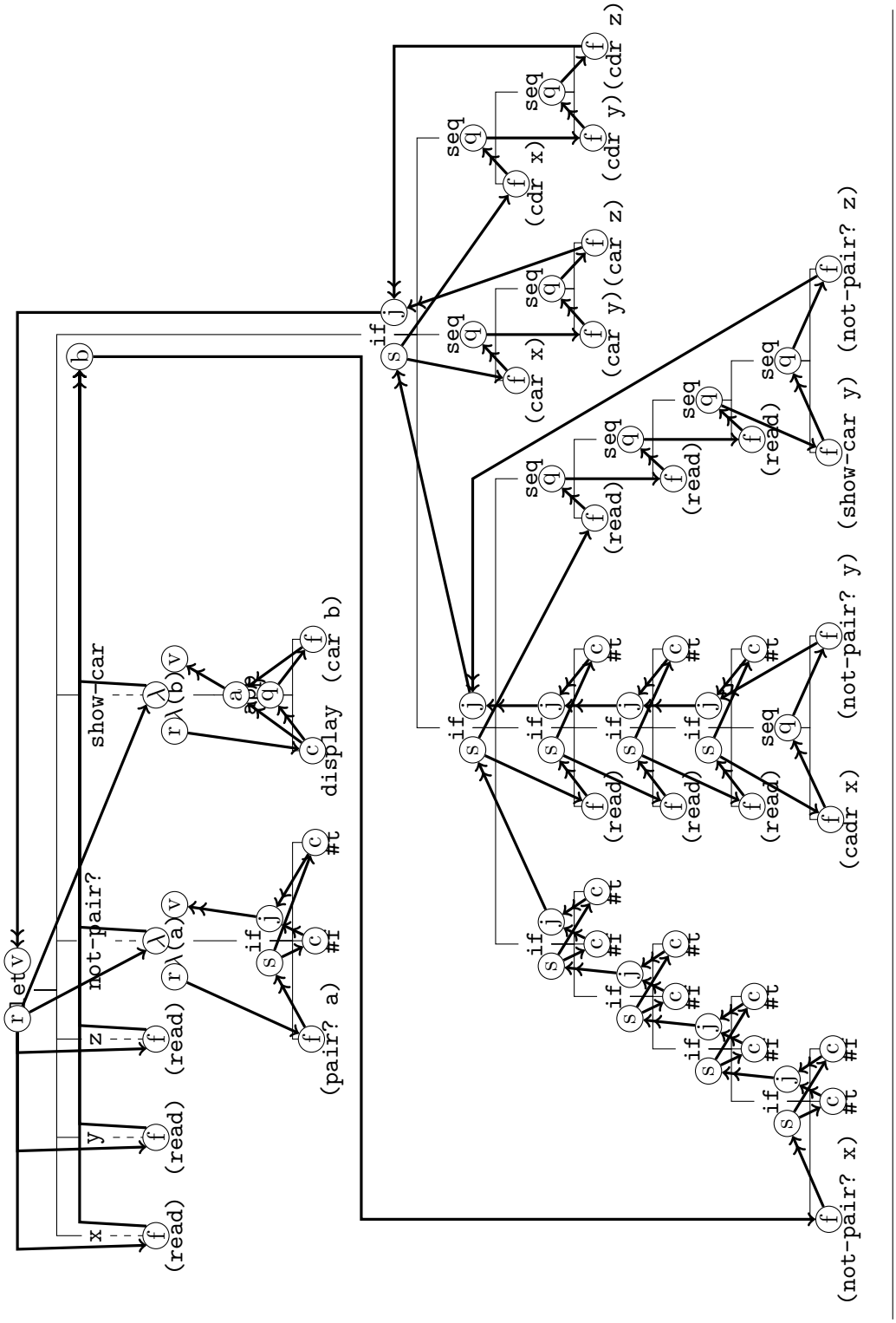


Figure 5.3.: Value flow

v: The result value of an expression. It is used both for the root expression and the body of each function. For a function body, it is used at function call sites to determine the return value of the function.

f: A function call. The edge going into the node determines whether the call is reachable. The edge going out of the function call contains the return value of the call. The nodes presented here are simplified to avoid complicating the figures.

c: A constant. The edge going into the node determines whether the constant is reachable and the edge going out of the node contains the value returned by the constant.

λ: A function. The edge going into the node determines the reachability of the function construction, and the edge going out of the node contains the closure constructed by the function. The body of the function is not connected to this node as the reachability and result of the function body interact with function calls and not function construction.

a: A function application. The two edges going into the application contain the function and argument for the function application. The edge going out is the return value of the function application.

b: The reachability of the body of a `let`. It is $\top$ if and only if the right-hand side of every binding in the `let` returns a value that is not $\bot$.

s: The reachability of the consequent and alternative of an `if`. The edge into the node is the result value of the test part of the `if`. The two edges going out of the node are the reachability of the consequent and alternative of the `if`. If the test part returns any true values, the consequent is reachable. If the test part returns any false values, the alternative is reachable.

j: The result value of an `if`. The two edges going into the node contain the result values of the consequent and alternative of the `if`. The result value of the `if` is computed by joining these values with $\sqcup$. The edge going out of the node is the result value of the `if`.

q: The reachability of the second expression of a `seq`. The edge going into the node contains the result value of the first expression of the `seq`. The edge going out of the node determines whether the second expression of the `seq` is reachable. It is reachable if and only if the first expression returns a value that is not $\bot$.

For each occurrence of a particular type of expression, the same set of nodes is created. For example, each `if` has both `s` and `j` nodes, and each `seq` as one `q` node. Thus, the size of the flow graph for values is linear in the size of the abstract syntax tree.

5.4. Context cache

Figure 5.4 shows the complete cache of contextual skipping functions as described in Section 4.4. The figure shows the *jump* edges of the Myers stacks that traverse each path of the abstract syntax tree. The *next* edges are omitted from the diagram. There is a *next* edge for each single-layer context in the abstract syntax tree.

With the exception of the root, each expression has exactly one *jump* edge coming out of it. Thus, the number of *jump* edges is linear in the size of the abstract syntax tree. Additionally, each *jump* edge contains either a single-layer context or is comprised of a single-layer context and two other *jump* edges. For example, the *jump* edge from `(not-pair? x)` up to the root is comprised of a single-layer context across the `if` that is the parent of `(not-pair? x)`, the *jump* edge from that `if` up and across the `if` that is the fourth ancestor of `(not-pair? x)`, and the *jump* edge from the fourth ancestor of `(not-pair? x)` up and across the root.

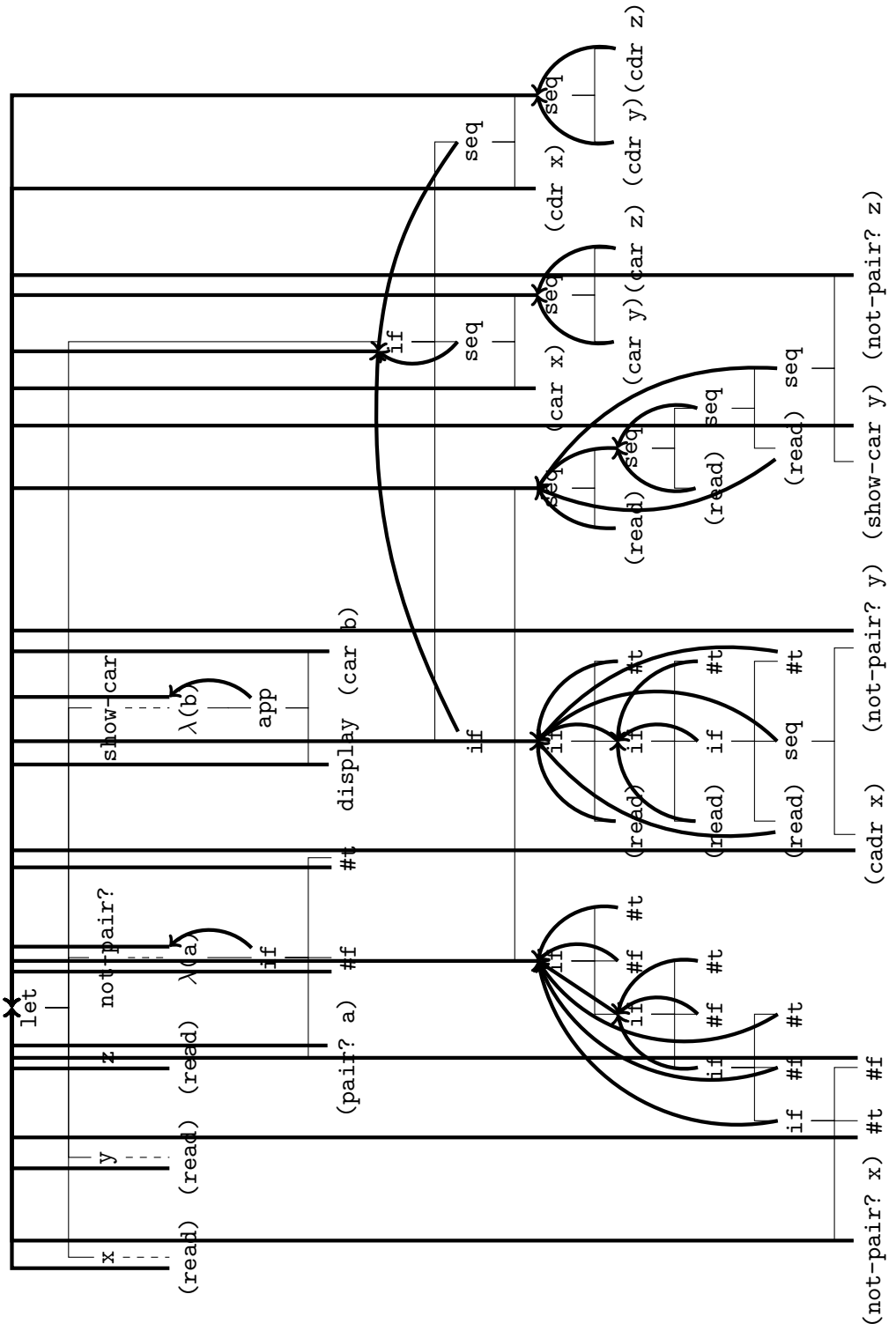


Figure 5.4.: Cache of contexts

These properties, which are guaranteed by the Myers stack construction, ensure that the cache both takes linear time to construct and only logarithmic time to update.

5.5. Variables

The parts of the flow graph for values and the contextual skipping functions, are variable independent. What remains is to define the flow graphs for each variable.

5.5.1. Flow graph for `x`

The variable `x` is referenced in both `(not-pair? x)` and `(show-car x)`. The first is deeply nested in the test part of a stack of `if` expressions. The second part is deeply nested in the consequent part of a stack of `if` expressions. The type information about `x` when the first stack returns true needs to be computed and used by the reference to `x` in `(show-car x)` in the second stack.

In order to do this efficiently, the first step is to calculate which contexts should be skipped for `x` based on the selection process described in Section 4.3. The lowest common ancestors that are selected for `x` are circled in Figure 5.5. The thick-lined arrows mark the contexts that are selected for skipping. The lower of the two circled `if` nodes contains two references to `x`. This means that Theorem 4.3 does not apply to it and it cannot be skipped. Once this `if` is selected, the rest of the context selection algorithm treats it opaquely and counts it as a single reference to `x`. Thus, the upper of the two circled `if` expressions is selected because it contains both the lower `if`, which counts for one reference, and `(car x)`, which counts for a second reference. The skips that are selected by this process go from the references to `x` up to the circled abstract syntax tree nodes. Normally, there would also be a context selected for skipping from the lower `if` to the upper `if`, but because they are adjacent, that context is empty and thus is omitted.

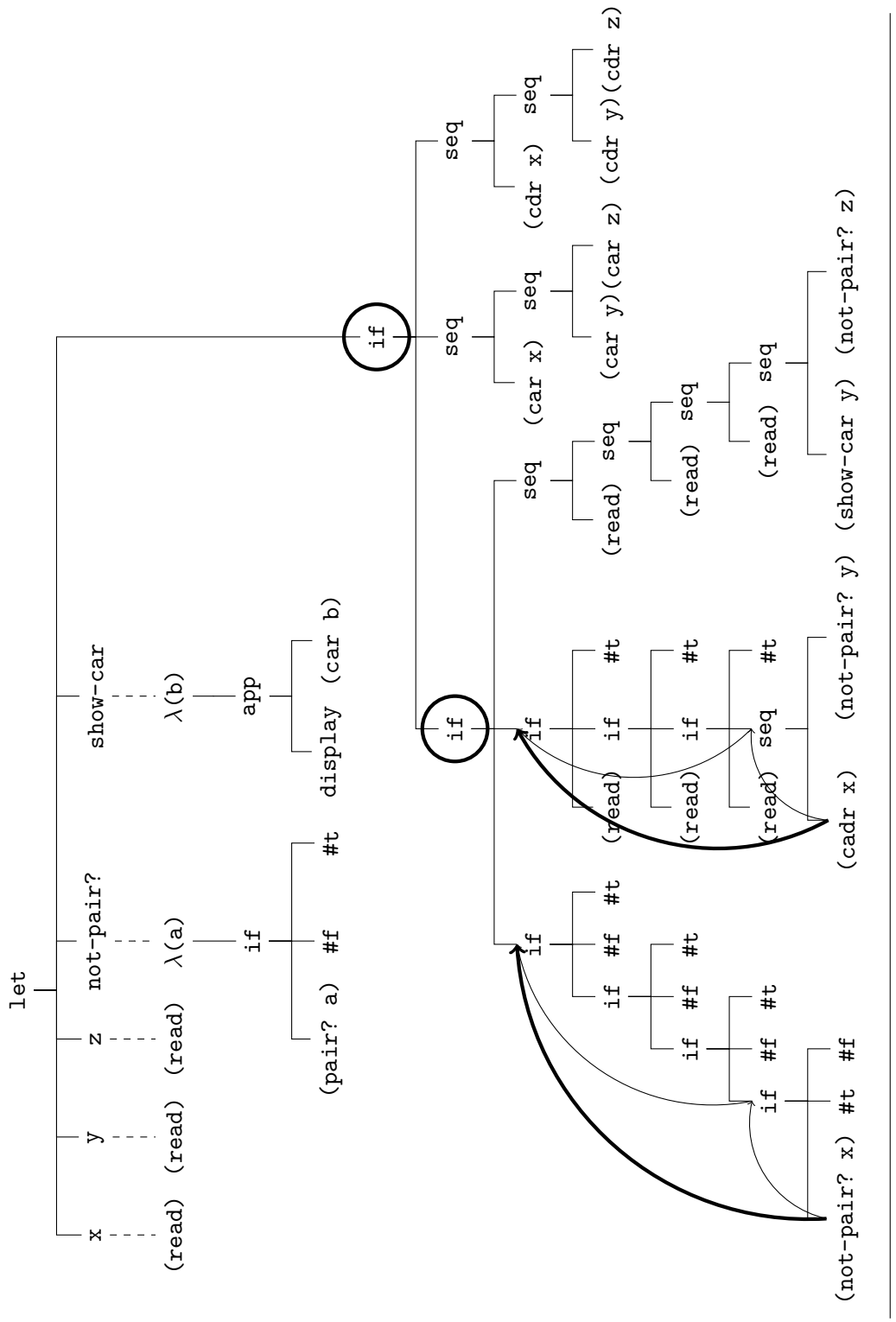


Figure 5.5.: Context skipping for x

To compute the skipping function for each of the selected skipping contexts, the appropriate contexts from the skipping function cache are composed. The thin lined arrows in Figure 5.5 show the contextual skipping functions that are taken from the skipping function cache and composed to form the selected contexts. These cached skips may in turn be composed of other skips in the cache. In the worse case, the skips for the selected contexts is composed from logarithmically many skips from the cache.

Once the selected skips are constructed, the type information for `x` flows through the program using those contexts, as shown in Figure 5.6. Recall that by Theorem 4.6, variables can freely skip down an expression, so the value for `x` moves directly from its binding site to `(not-pair? x)`. Then the skipping function is used to move `x` from `(not-pair? x)` up to the fourth ancestor of `(not-pair? x)`. If decomposed fully, this skipping function is composed of four single-layer contextual skipping functions corresponding to the `if` expressions being skipped. Three of theses expressions return `#f` in the consequent and `#t` in the alternative and thus each swap the values for the true and false cases. One of these expressions returns `#t` in the consequent and `#f` in the alternative and thus passes the values for the true and false cases through unchanged. Hence, the composition of these four skips is equivalent to a single swap. This swap, when applied to the type information about `x` coming from `(not-pair? x)`, results in the information that when the fourth ancestor of `(not-pair? x)` returns true, `x` is a pair and when it returns false, `x` is not a pair. This result is stored in the node marked $\mathcal{V}$.

Next, the fifth ancestor of `(not-pair? x)` is one of the common ancestors marked for `x`, so the usual constraint rules are used there instead of the skipping functions. Thus, the value from the $\mathcal{V}$ node moves to the `s` node, and the `s` node splits the type information about `x` into the true and false parts. The true part is sent to the consequent, and the false part is sent to the alternative. In this case, that means that `x` is known to be a pair in the consequent and a non-pair in the alternative.

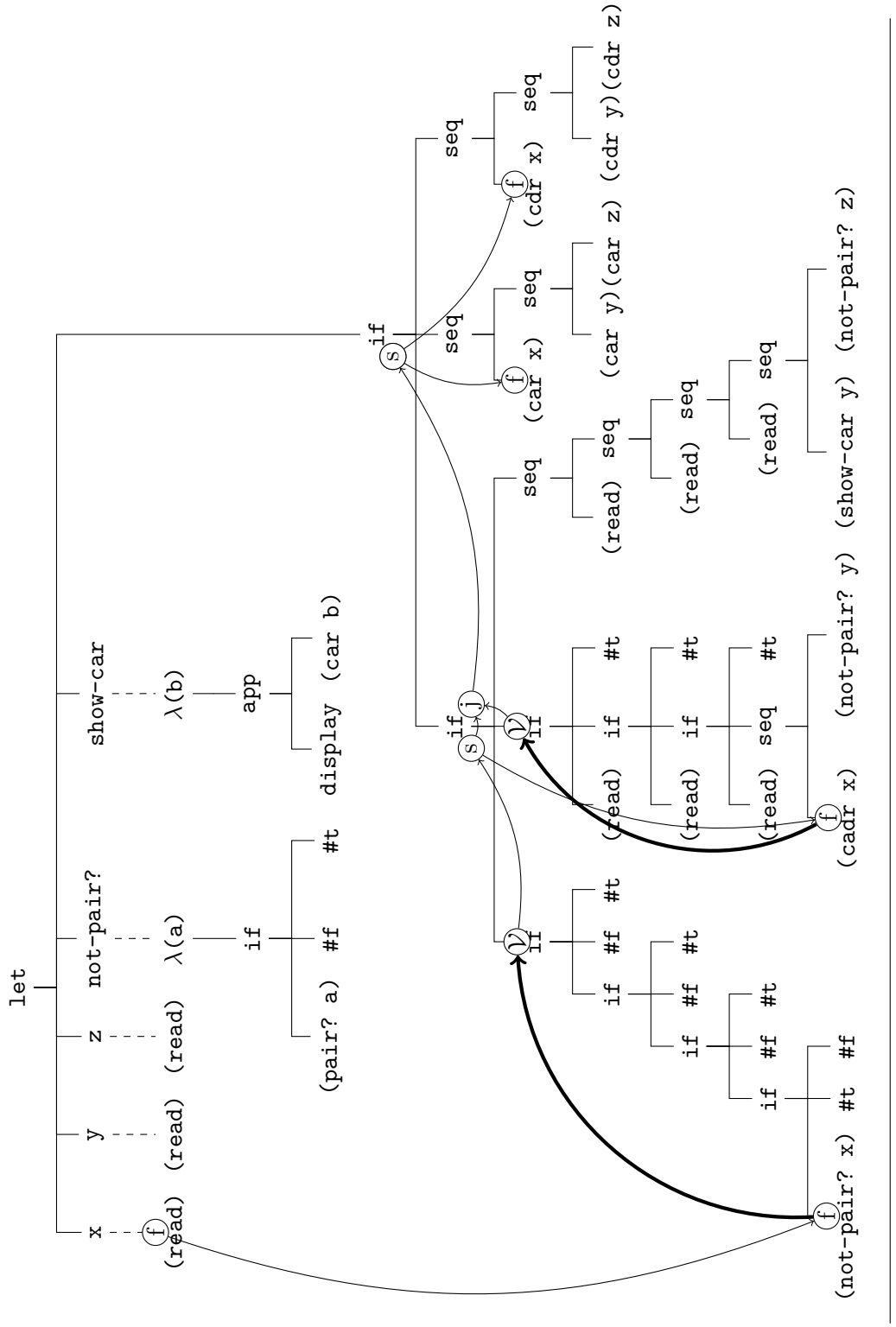


Figure 5.6.: Flow graph for x

In the consequent, the type information for `x` flows directly down to `(cadr x)` per Theorem 4.6. Then it flows back up again using the selected skipping function. As before, the constraint rules are used for manipulating the type information at the fifth ancestor of `(not-pair? x)`. The `j` node joins the information from the consequent, which indicates that `x` is a pair in both the true and false cases, with the information from the alternative. Since the alternative has no references to `x`, that information is taken directly from what the `s` node sent to the alternative. In this case, it says that in the alternative `x` is a non-pair for both the true and false cases. Joining information from both branches results in the knowledge that, in the true case, `x` may be a pair due to the consequent or a non-pair due to the alternative. Likewise, in the false case, `x` may or may not be a pair.

The flow of information proceeds in a similar manner on through the sixth ancestor of `(not-pair? x)` and to the `(car x)` and `(cdr x)`.

5.5.2. Flow graph for `y`

The process for `y` is similar to the process for `x`, and is only slightly different. As with `x`, the first step to processing `y` is to select skipping contexts and construct skipping functions for those contexts. Figure 5.7 shows the equivalent of Figure 5.5 but for `y` instead of `x`. The process and notational conventions are identical to the process and conventions for `y`.

Once the selected skipping contexts are constructed, the type information for `y` flows through the program using those contexts as shown in Figure 5.8. As with `x`, the value of `y` flows directly down from its binding site. However, `y` is in both the consequent and alternative of the lower of the two circled `if` expressions instead of the test and consequent. There is no sequential ordering between the consequent and alternative, so `y` flows from its binding site directly to both the consequent and alternative instead of first to the consequent and then to the alternative. Thus, on entry to both the consequent and alternative, `y` remains at $\top$.

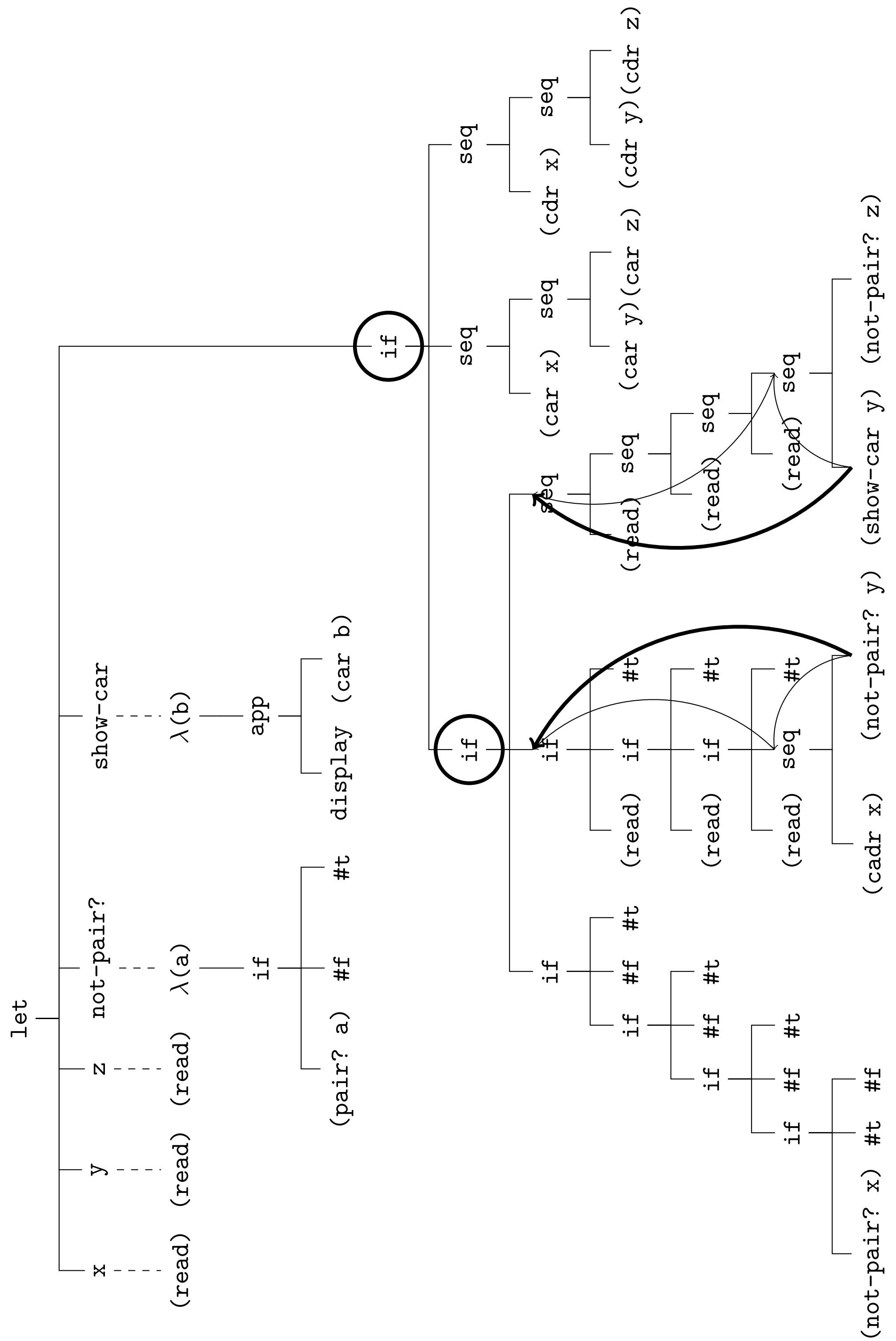


Figure 5.7.: Context skipping for y

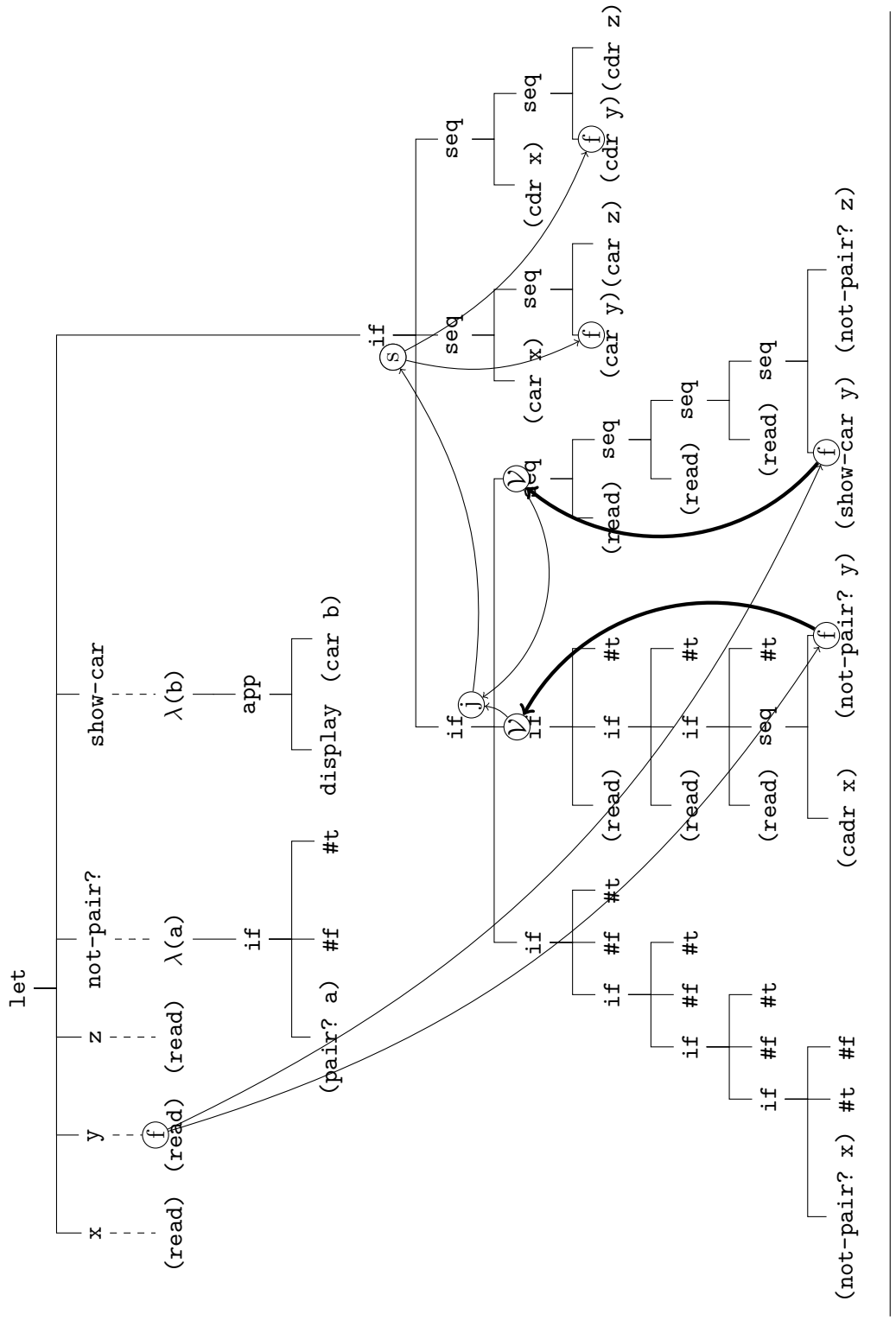


Figure 5.8.: Flow graph for y

After `(not-pair? y)` in the consequent, `y` is known to be a pair if the expression returns false and a non-pair if the expression returns true. The skipping function for the context that goes from `(not-pair? y)` up to and over the fourth ancestor of `(not-pair? y)` modifies this. The part that skips over the `seq` behaves as the identity. For the intervening `if`, the skipping function accounts for the possibility that each of the `(read)` calls in the `if` expressions may return either true or false. If `(read)` returns false, the alternative of the corresponding `if` returns `#t`. Thus, when one of these `if` returns a true value, it could be from either `not-pair?` or `#t`. On the other hand, when one of these `if` returns a false value it can only be from `not-pair?`. The end result is that, at the $\mathcal{V}$ node, nothing is known about `y` if the corresponding `if` returns true, but `y` is known to be a pair if it returns false.[1]

A similar process occurs for `(show-car y)`. It is, however, slightly simpler, since `show-car` unconditionally restricts `y` to be a pair. Even if it did not, the skipping function moving out of the first expression of a `seq` erases all conditional information by joining the true and false information using the $\sqcup$ operator. In any case, the information for `y` at the $\mathcal{V}$ node records that `y` is always a pair when exiting the corresponding `if`.

At this point, when exiting the fourth ancestor of `(not-pair? y)`, `y` is known to be a pair if the fourth ancestor returns false, but nothing is known if the fourth ancestor returns true. When exiting the fourth ancestor of `(show-car y)`, `y` is known to always be a pair. The constraint rules take this information to compute the value of `y` when exiting the fifth ancestor by joining the two results. Thus, when exiting the fifth ancestor, `y` is known to be a pair in the false case, but nothing is known in the true case.

This information flows to the `s` node, which sends the information for the true case to the consequent and the information for the false case to the alternative. Thus, at `(car y)`,

[1] A similar contextual skipping function applies to `x`, but in that case `x` is already known to be a pair due to the `(not-pair? x)`, so that detail was omitted when describing the information flow for `x`.

nothing is known about `y`, and the type check is still needed. At `(cdr y)`, `y` is known to be a pair and no type check is needed.

5.5.3. Flow graph for `z`

As before, the first step to processing `z` is to select skipping contexts and construct skipping functions for those contexts. The result of this is shown in Figure 5.9. The circled `if` is the lowest common ancestor of any two occurrences of `z`. The selected skipping context in this case is constructed from three other skipping contexts. The two with thin solid lines come from *jump* edges in the cache. The one with dashed lines comes from a *next* edge in the cache.

Once the selected skipping contexts are constructed, the type information for `z` flows through the program using those contexts as shown in Figure 5.10. The process is almost identical to the process for `x` and `y` with one major difference: the contextual skipping function for flowing from `(not-pair? z)` up to and over the fifth ancestor of `(not-pair? z)` must account for the possibility that the `if` may take the consequent instead of the alternative. In the case of `x` and `y`, this was accounted for because the constraint rules were used instead of the contextual skipping function. In the case of `z`, however, the contextual skipping function must account for this. This is why in Figure 4.2 the contextual skipping function takes three input types rather than just two. The first two types are for when the expression returns true and false respectively. The third type is the type when entering the expression and handles cases like this. In this case the contextual skipping function records the fact that, since the consequent returns both true and false values, whatever type `z` has when entering this expression is potentially part of the output of this expression in both the true and false cases. If the fifth ancestor returned only true or only false values, the type of `z` when entering the expression would be part of only the true or false cases respectively. Though this is a trivial observation in this example, when these situations

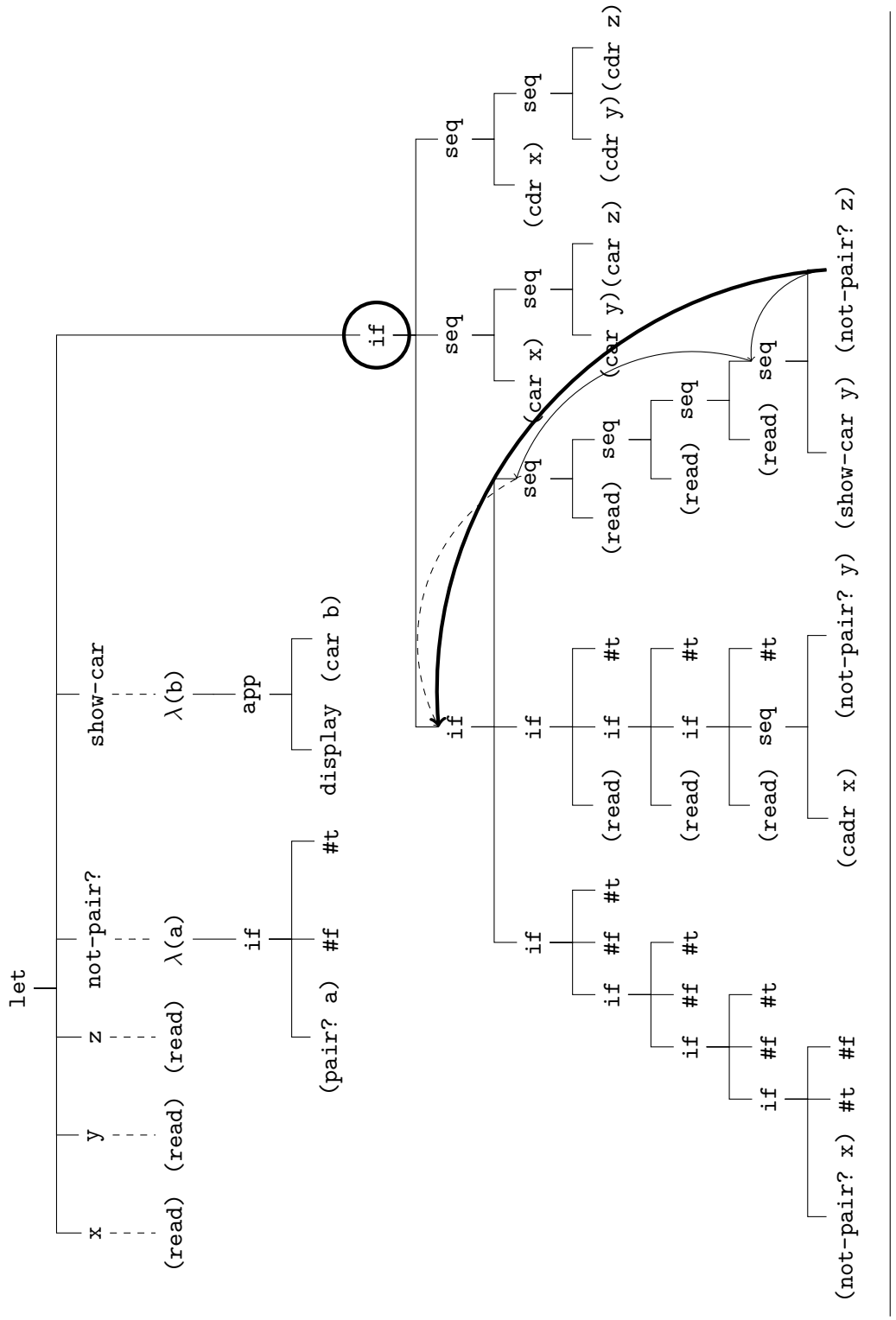


Figure 5.9.: Context skipping for z

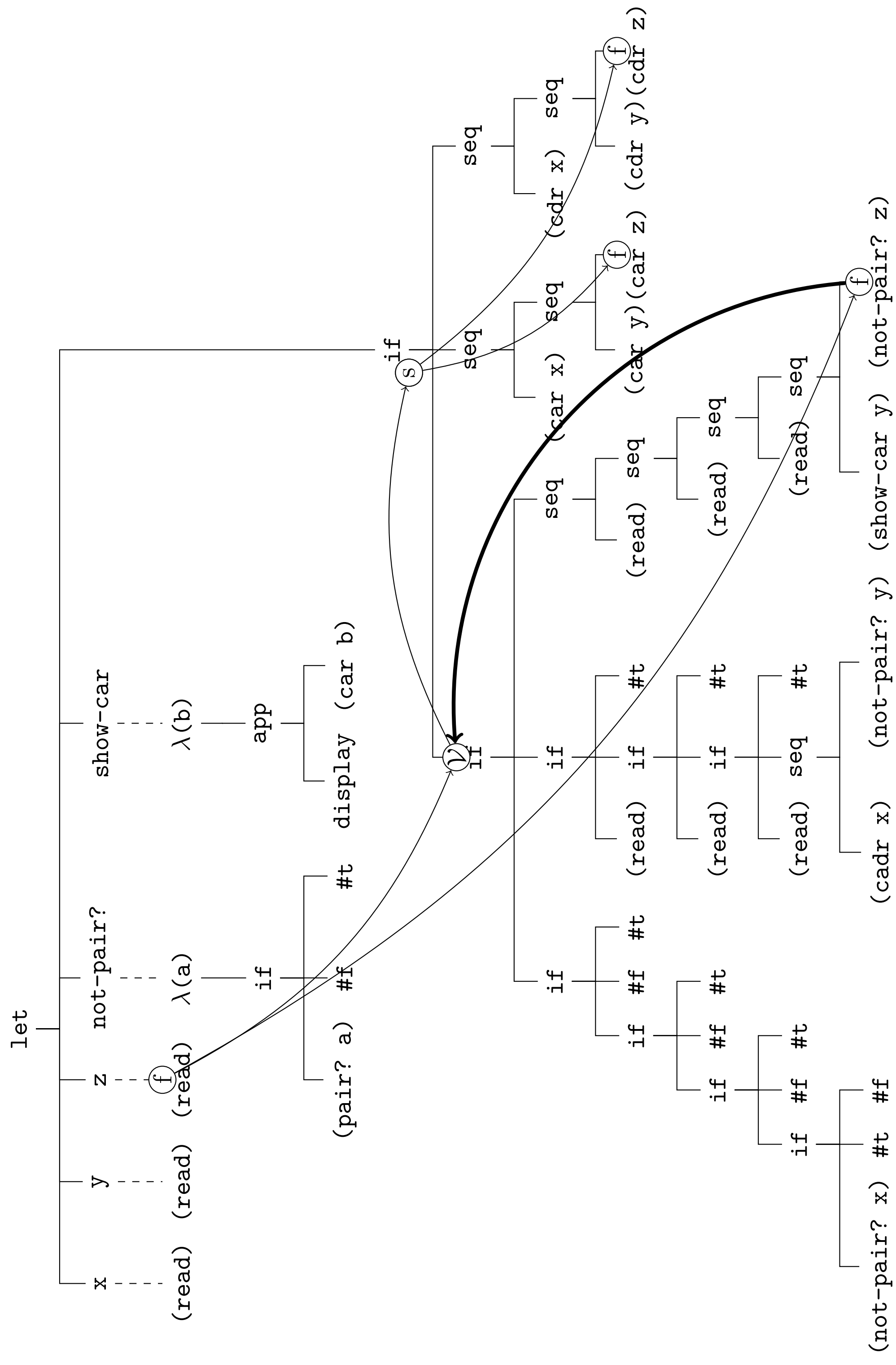


Figure 5.10.: Flow graph for z

are in the middle of a larger selected skipping context, it is important to account for this possible execution path.

Aside from this subtlety, the flow graph for `z` behaves similarly to the flow graphs for `x` and `y`.

5.5.4. Other variables

Flow graphs for `not-pair?`, `show-car`, `a`, and `b` are constructed following exactly the same process as for `x`, `y`, and `z`. They are omitted here as they do not illustrate anything noteworthy.

5.6. Result

Once the flow graph is constructed, it is processed using the standard technique described in Section 2.2 for evaluating a flow graph. Once the flow graph stabilizes, the following is information is learned. All calls to `not-pair?` and `show-car` are known calls to their respective functions. The `(cadr x)` expression always has a pair value for `x` and does not need a pair check, but `(car x)` may or may not be called on a pair value and thus needs a pair check. The `(car y)` expression always has a non-pair value for `y`, but `(cdr y)` may or may not be called on a pair value and thus needs a pair check. Both `(car z)` and `(cdr z)` need pair checks, since `z` may or may not be a pair at those points.

From one perspective, it might appear that the analysis did poorly on this example as it was able to determine non-top types only for `(cadr x)` and `(car y)`. From another perspective, however, the analysis performed perfectly. The places where the analysis cannot determine the exact results are due to the semantics of the program and not due to approximations made by the analysis. Depending on the return values of each of the `read` calls, the program actually could have either pair or non-pair values in every place

that the analysis could not determine a precise value. In this sense, the analysis performs excellently. The benchmarks in Chapter 7 study the performance of the analysis in more detail.

6. Implementation

The CFA algorithm described in Chapter 4 has been implemented as an experimental optimization for the Chez Scheme [Dybvig, 2010] compiler. It is used to perform type recovery and justify the elimination of run-time type checks. The implementation supports the full Scheme language and successfully compiles and runs both Chez Scheme itself and the entire Chez Scheme test suite without errors.

6.1. Implementation structure

To implement type recovery, a post-processing pass is added after the CFA pass. The post-processing pass uses the type information gathered during the CFA pass to determine where run-time type checks are unnecessary. Primitive calls where some or all of the run-time type checks are unnecessary are replaced by an "unsafe" variant that does not perform the unnecessary run-time type checks. For instance, `(car x)` is replaced by `(unsafe-car x)` when `x` is determined to be a pair. If a primitive makes multiple run-time type checks and only some of those type checks can be omitted, then a "semi-unsafe" variant is used. These cases arise when a primitive does more than one run-time type check or when the checks involve information not tracked by the analysis. For example, a vector range check cannot be eliminated because the analysis does not track the lengths of vectors. Another example is when the analysis determines that the vector argument of a `vector-ref` is always a vector but not that the index argument is always a nonnegative exact integer.

6.2. Implementation notes

The implementation handles a variety of language constructs and features that are not described in Chapter 4. Among these are mutable variables and the unspecified order of evaluation for function call arguments and `let` bindings.

A mutable variable's type can change between where type information is recovered and it is used. For instance, an intervening function call could arbitrarily mutate the variable and invalidate what is learned.[1] Thus, for mutable variables, the implementation gathers only constructive information.

The unspecified order of evaluation for function-call arguments and `let` bindings can be handled by choosing a fixed evaluation order prior to this analysis. At present, however, the decision is made later in the compiler, during register allocation. The analysis therefore processes function-call arguments independently, as it does for the branches of an `if`. While the resulting environments are unioned for `if`, they are intersected for function-call arguments. The bindings of `let` are handled similarly.

6.3. Order of evaluation

As defined in Chapter 3 always assumes a left to right order of evaluation between function arguments. However, because the Chez Scheme compiler determines order of evaluation after the analysis runs, the analysis implemented for Chez Scheme does not assume a particular order of evaluation. For example in an expression like `(f (car x) (cdr x))`, the analysis is agnostic about whether `car` or `cdr` is evaluated first. This has the advantage of giving later passes of the compiler the freedom to choose the order of evaluation, but also has the disadvantage of preventing the analysis from eliminating the implicit type checks

[1] This issue arises only in higher-order languages. The analysis can process restrictive information for mutable variables in first-order languages, including, for example, the output language of a closure-conversion pass in a typical compiler for a higher-order language.

$$\begin{aligned}
\text{Expressions:} \quad & e \in \mathit{Exp} = \ldots \mid \mathit{let}\ x = e\ \mathit{in}\ e \mid \mathit{letrec}\ x = e\ \mathit{in}\ e \\
\text{Contexts:} \quad & C \in \mathit{Ctxt} = \ldots \mid (\mathit{let}\ x = \Box\ \mathit{in}\ e) \mid (\mathit{let}\ x = e\ \mathit{in}\ \Box) \\
& \mid (\mathit{letrec}\ x = \Box\ \mathit{in}\ e) \mid (\mathit{letrec}\ x = e\ \mathit{in}\ \Box)
\end{aligned}$$

$$\frac{[\![\mathit{let}\ x = e_0\ \mathit{in}\ e_1]\!]_{in} \sqsupseteq \langle \hat{r},\, \hat{\rho} \rangle}{[\![e_0]\!]_{in} \sqsupseteq \langle \hat{r},\, \hat{\rho} \rangle}\ \text{LET}_{in}$$

$$\frac{[\![e_0]\!]_{out} \sqsupseteq \langle \hat{v}_0,\, \hat{\rho}_t,\, \hat{\rho}_f \rangle \qquad \mathbb{K}(e_0) = (\mathit{let}\ x = \Box\ \mathit{in}\ e_1)}{[\![e_1]\!]_{in} \sqsupseteq \langle \top,\, (\hat{\rho}_t \sqcup \hat{\rho}_f)\,[x \mapsto \hat{v}_0] \rangle}\ \text{LET}_{mid}$$

$$\frac{}{[\![\mathit{let}\ x = e_0\ \mathit{in}\ e_1]\!]_{out} \sqsupseteq [\![e_1]\!]_{out}}\ \text{LET}_{out}$$

$$\frac{[\![\mathit{letrec}\ x = e_0\ \mathit{in}\ e_1]\!]_{in} \sqsupseteq \langle \hat{r},\, \hat{\rho} \rangle}{[\![e_0]\!]_{in} \sqsupseteq \langle \hat{r},\, \hat{\rho} \rangle}\ \text{LETREC}_{in}$$

$$\frac{[\![e_0]\!]_{out} \sqsupseteq \langle \hat{v}_0,\, \hat{\rho}_t,\, \hat{\rho}_f \rangle \qquad \mathbb{K}(e_0) = (\mathit{letrec}\ x = \Box\ \mathit{in}\ e_1)}{[\![e_1]\!]_{in} \sqsupseteq \langle \top,\, (\hat{\rho}_t \sqcup \hat{\rho}_f)\,[x \mapsto \hat{v}_0] \rangle \quad [\![e_0]\!]_{in} \sqsupseteq \langle \top,\, (\hat{\rho}_t \sqcup \hat{\rho}_f)\,[x \mapsto \hat{v}_0] \rangle}\ \text{LETREC}_{mid}$$

$$\frac{}{[\![\mathit{letrec}\ x = e_0\ \mathit{in}\ e_1]\!]_{out} \sqsupseteq [\![e_1]\!]_{out}}\ \text{LETREC}_{out}$$

Figure 6.1.: Rules for `let` and `letrec`

in either `car` or `cdr`. With a fixed order of evaluation on the other hand, say left to right, the `car` is certain to evaluate before the `cdr`. Thus, the `cdr` can only be reached if `x` is a pair, and the implicit pair check in the `cdr` can be eliminated.

Arbitrarily fixing an order of evaluation is quite easy, but it would be far better to chose an evaluation order based for each expression based on how many type checks the particular order allows the analysis to eliminate. How to achieve this is a question that deserves further research.

6.4. Local binding

Both `let` and `letrec` are implemented via the obvious extensions. The flow rules for these are shown in Figure 6.1.

Note that, `(let ([x` e_1`]) ` e_2`)` learns more information than the usually equivalent expression `((lambda (x)` e_2`)` e_1`)`. With `let`, the analysis uses observational information from e_1 and passes it to e_2. With `lambda`, however, the environment passed to e_2 is the one from the construction of the function by `lambda`. If a function call evaluates e_1 before `(lambda (x)` e_2`)`, then the `lambda` and thus e_2 see the observational information from e_1. However, with an indeterminate order of evaluation for application, e_2 cannot use information learned from e_1 as the `lambda` may be evaluated before e_1.

The implementation of this analysis includes a preprocessing pass that converts from the `lambda` form to the equivalent `let` form to ensure that as much information as possible is learned from this expression. This is equivalent to fixing the order of evaluation to be from right to left for these expressions, but avoids needing a special case in the code for function applications.

When the right-hand sides of a `let` do not mention any of the variables bound by the `let`, it is nearly equivalent to a `letrec`. The only difference is the handling of continuations. Thus, in order to avoid code redundancies between `let` and `letrec`, the implementation converts all `let` expressions into `letrec` expressions. Since all variables are uniquely named (Section 2.1.1), none of the variables bound by the `let` occur in the right-hand sides and this is a valid transformation. A post-processing pass converts these `letrec` expressions back to `let` expressions [Waddell et al., 2005]. This expedient simplifies the implementation for research purposes, but in a full scale implementation it should be replaced with separate handling of `let` and `letrec` forms.

6.5. Lambda

Full Scheme includes a more powerful form of `lambda` than in Figure 2.1 or Figure 3.1. It includes both variable argument forms and `case-lambda` forms.

Variable argument forms The variable argument form in Scheme allows a function to take n or more arguments rather than exactly n arguments. In this case, the extra arguments are bundled into a list and bound to an extra parameter. If a function call passes $n + m$ arguments, the first n formal parameters are bound to the first n arguments as usual, and the final parameter is bound to a list of the last m arguments.

To handle these extra arguments, each function application must check how many arguments the function expects and whether it uses the variable argument form of `lambda`.

If the variable argument form is not used, processing is just as before. If the variable argument form is used, however, the analysis must consider three cases.

In the first case, the function requires more arguments than the application provides. The function call then triggers an error at run time and thus the analysis does not flow any values from the call to the formal parameters.

In the second case, the function requires the exact same number of arguments as the call provides. At run time, the extra parameter receives the empty list and thus the analysis adds the empty-list type to the input type of the extra parameter. It also checks whether the true and false output types of the extra parameter contain the empty-list type. If they do, the function can return to the function application. Otherwise, it can not.

In the final case, the function requires fewer arguments than the application provides. At run time the extra parameter will receive a non-empty list and thus the analysis adds the pair type to the input type of the extra parameter. Likewise, it also checks whether the true and false output types of the extra parameter contain the pair type. If they do, the function can return to the function application. Otherwise, it can not.

Case-lambda The `case-lambda` form generalizes of the usual `lambda` form by including multiple clauses. Each clause contains both formal parameters and a body. When the function is called, which clause executes depends on the number of arguments provided by the function call. Each clause is sequentially checked, and the first clause is used that either does not use the variable argument form and expects the exact number of arguments that the function application provides or that does use the variable argument form and requires fewer arguments than the function application provides. Once the clause is selected, execution proceeds as usual.

In order to handle this, the analysis takes advantage of the fact that, given the number of arguments provided by the function call, it can call at most one of the causes. Then it is simply a matter of efficiently determining which clause of the `case-lambda` a particular function call uses. The analysis does via a preprocessing pass that builds a table for each `case-lambda`. The ith element of this table points to the clause that should be invoked by a function call with i arguments. The table also stores a default clause to use when there are more arguments in a function application than there are entries in the table. Thus, finding the correct clause of a `case-lambda` takes $O(1)$ time. Computing this table in only $O(n \log n)$ time takes a careful engineering but is fairly straightforward.

6.6. Smart primitives

In the general case, functions like `+` and `call/cc` provide very little information about their return values. Without knowing anything more about their arguments, the best we can say is that `+` returns some sort of number. We know nothing about the return value of `call/cc`. However, if a particular invocation of `+` always passes integer values in, the return value will be an integer value. Likewise, the result of `(call/cc f)` will be the return value of `f` joined with the arguments of the continuation that is passed to `f`. Both

of these more general situations can be handled by "smart" primitives that contain code to compute return types rather than having a static return type.

For example, when the type of an argument passed to an occurrence of `+` is updated, the code for `+` calculates the potential return types and updates its return type appropriately. Thus, if an occurrence of `+` is passed only positive integers, the return type will be positive integers. On the other hand if the arguments are floating-point numbers, the return type will be the type of floating-point numbers.

The same technique works for `call/cc`. Each occurrence of `call/cc` creates a function object that represents the continuation that is passed to the argument of `call/cc`. For example, consider the continuation passed to `f` in `(call/cc f)`. When the function type representing this continuation is applied to an argument, the input types of its argument are updated. This in turn notifies `call/cc`, which updates its return value to contain the types that could be passed to the continuation.

6.7. Mutable variables

When a variable is mutated, it is not a safe approximation to ignore the mutation, since the value might become a value not approximated by the original value. For example, in Example 6.1, ignoring the `set!` inside `f` would lead to the conclusion that `x` is a pair at the call to `car`.

```
(let ([x (read)])                                          (6.1)
  (let ([f (lambda () (set! x 2))])
    (car x)
    (f)
    (cdr x)))
```

The analysis assumes that information from inside function calls such as `(f)` can be safely ignored. When the variable is immutable, the function cannot change the value, so all information about a variable before the call is still valid after the call. The worst that

happens is the analysis computes a less precise value than if it used information from inside function calls. However, this is not the case when the function can change the value. In this example, `f` changes the type of `x` from being a pair to being a number.

To deal with this the analysis adjusts the value seen at the binding site of a mutable variable to include all values that `set!` stores into the variable. It then ignores all restrictive information about mutable variables. Thus, the analysis collects only constructive and not restrictive information about mutable variables.

This problem is well known in the static analysis domain as the *strong update problem* [Lhoták and Chung, 2011] and is not a problem particular to the analysis presented in this dissertation. Standard solutions to the strong update problem or using *known call* information [Adams et al., 1986, Appel and Jim, 1989] may help deal with this, but such solutions are beyond the scope of this dissertation.

Note that a solution to the strong update problem may lead to improvements in the precision of the analysis even when dealing with immutable variables. For example, consider the following expression.

```
(let ([x (read)])                                                    (6.2)
  (let ([f (lambda () (car x))])
    (f)
    (cdr x)))
```

The analysis described here does not detect that `x` must be a pair after exiting `f`. Thus, it misses the opportunity to eliminate the type check in `(cdr x)`. In a sense, the `car` assigns to the type that the analysis knows for `x` by restricting the type of `x` to only pairs. Thus, a method of accounting for strong update is likely to generalize to also handle cases like this.

7. Benchmarks

7.1. Effectiveness

The effectiveness of the type-recovery algorithm was tested on a standard set of Scheme benchmarks [Clinger, 2008]. Each test was run both with type recovery enabled and with type recovery disabled. The number of type checks performed in these two cases were then compared with each other. The tests look only at type checks caused by pair and vector primitives such as `car`, `cdr`, `cadr`, `vector-ref`, or `vector-set!`. An average of 69% of type checks are eliminated from the code, which results in 55% fewer type checks at run-time. These results were compared with a flow-insensitive version of this analysis, which performed significantly worse, eliminating 41% of run-time type checks. The results of the flow-sensitive sub-0CFA were also compared with the results of a flow-sensitive 0CFA. The 0CFA version performed about the same, also eliminating 55% of type checks at run time. A flow-insensitive 0CFA performed better than the flow-insensitive sub-0CFA, eliminating 48% of run-time type checks. Figure 7.1 compares the average percent of type checks eliminated for the flow-insensitive sub-0CFA, flow-insensitive 0CFA, flow-sensitive sub-0CFA, flow-sensitive 0CFA. Figure 7.2 and Figure 7.3 give the percent of checks eliminated for each individual program.

While these results are promising, they are not necessarily a good predictor for how well type-recovery performs in general. Using the same set of tests, and counting only the checks

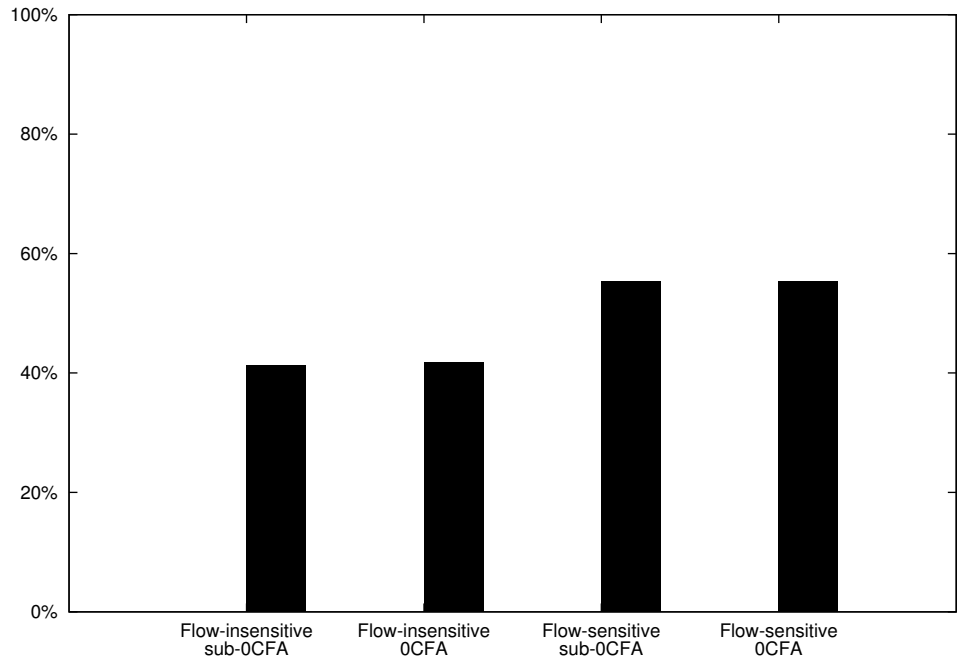


Figure 7.1.: Average percent of type checks removed

made by pair primitives, only 36% of checks are eliminated, resulting in the elimination 30% of run-time checks. Pairs are a hard case for type-recovery. While vectors store a number of items, pairs store only two. At best, we ordinarily expect to see a call to `car` paired with a call to `cdr` and can eliminate only one of the two type checks. In operations such as `cadr` and `cddr` only the first of two pair checks can be eliminated. This is because the contents of pairs are not tracked by the analysis. It cannot determine that the `cdr` of a pair is also a pair. The nested pair must always be type checked. Also, with proper lists, it is common to use an explicit `null?` check to detect the end of the list. Unfortunately, the `null?` check does not eliminate the need for the `pair?` checks implicit in `car` and `cdr`, since it tells the analysis only that the value is not null. Beyond the difficulties in handling pairs, some opportunities for eliminating type checks are already handled by the source optimizer before getting to the type recovery analysis.

Although the analysis does not require the order of evaluation of `let` bindings and function-call arguments to be specified, type information learned in one argument or binding might be useful for eliminating a type check in another argument or binding. For instance, in `(f (car x) (cdr x))`, a specified evaluation order would allow the implicit pair check to be eliminated from one of the two argument expressions. To determine the impact of fixing the order of operations, the tests were run with both left-to-right and right-to-left evaluation orders. In both cases, although the benefit is significant in a few cases, the average number of type checks at run time improved by only a few percent.

These results are encouraging, and refinements of the analysis should further improve the optimization's effectiveness. The analysis currently treats all pairs and all vectors the same, although it could treat each occurrence of `cons` and `make-vector` in the source code as a separate element in the lattice analogously to the way it handles `lambda` expressions, and thus get more information about the contents of pairs and vectors.

7.2. Efficiency

Beyond the effectiveness of the analysis, its asymptotic behavior was verified and its speed measured by counting the number of source tree nodes on input to the type-recovery algorithm and measuring the time it takes for the algorithm to run. For this test, the same benchmarks were used as before, along with compilation units that comprise the Chez Scheme compiler. Figure 7.4 plots these times on a logarithmic scale along with linear (lower) and linear-log (upper) reference lines. The quantization at the lower end of the graph results from timer granularity. The graph shows that the processing times trend between the linear and worst-case linear-log lines as expected. The type recovery is also acceptably fast, handling 100,000 AST nodes (approximately 30,000 lines of code) in less than a second for the largest of the programs. Averaging over all of the programs, the

implementation handles about 281,500 AST nodes (approximately 100,000 lines of code) per second.

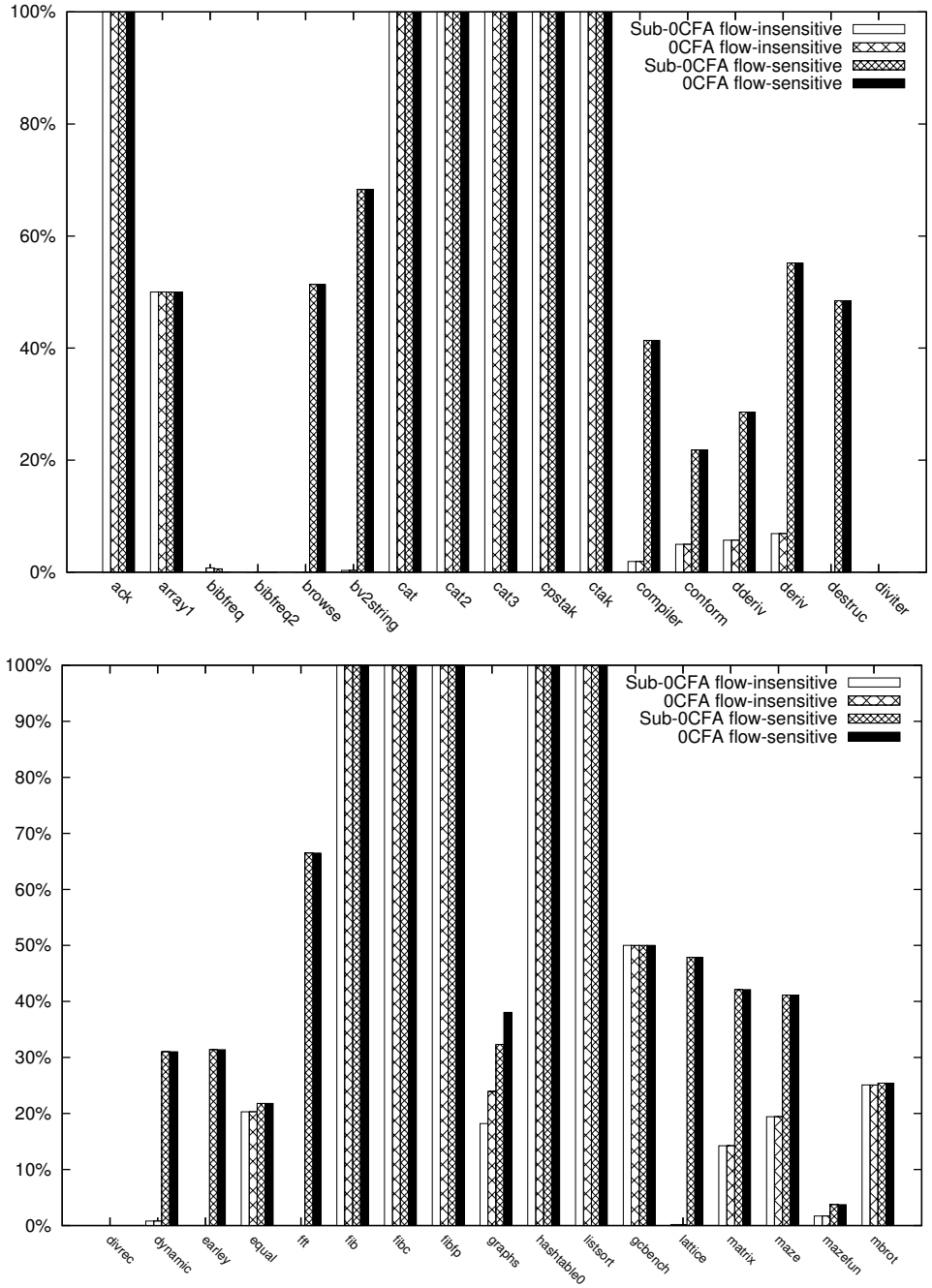


Figure 7.2.: Percent of type checks removed

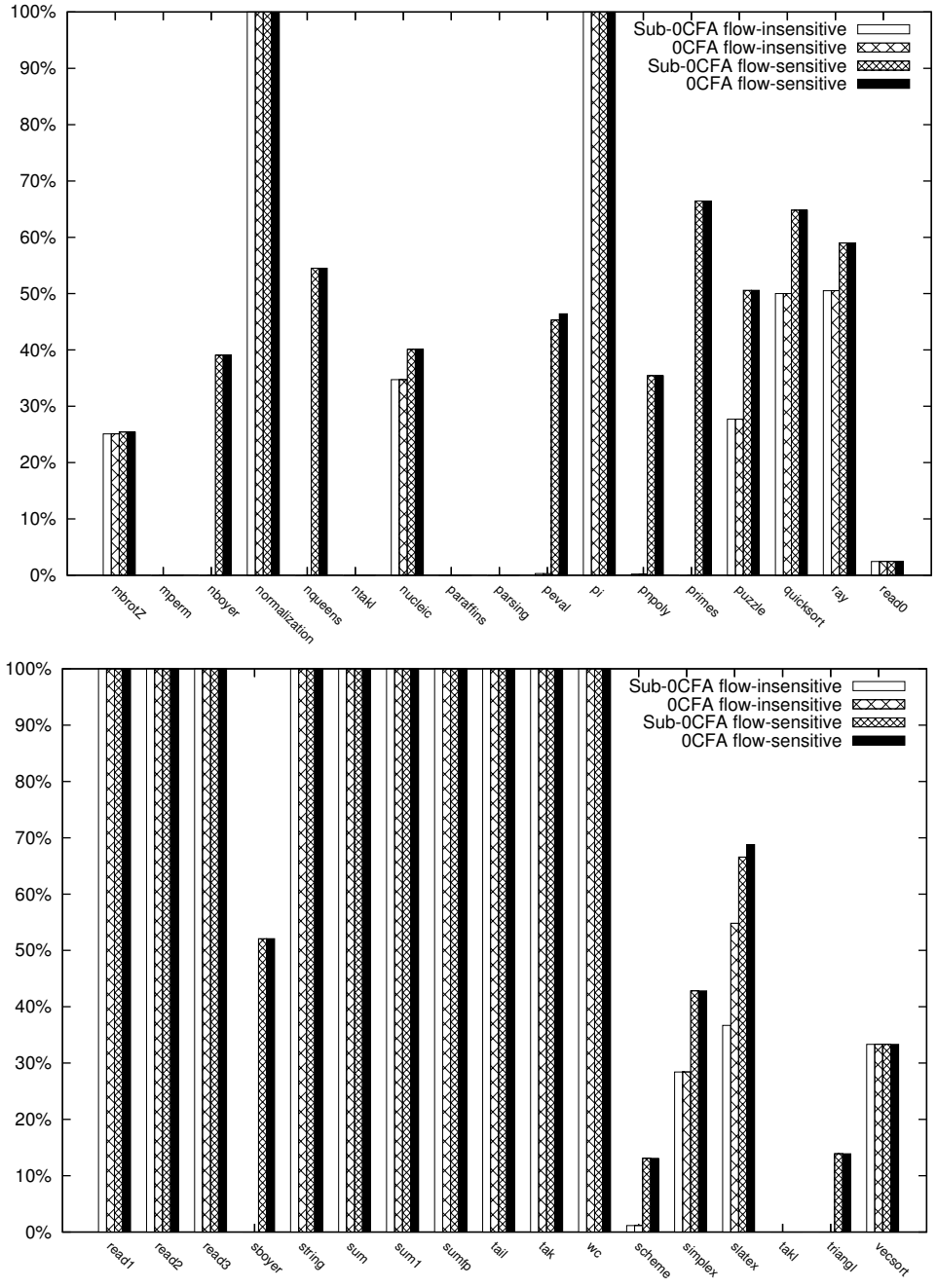


Figure 7.3.: Percent of type checks removed

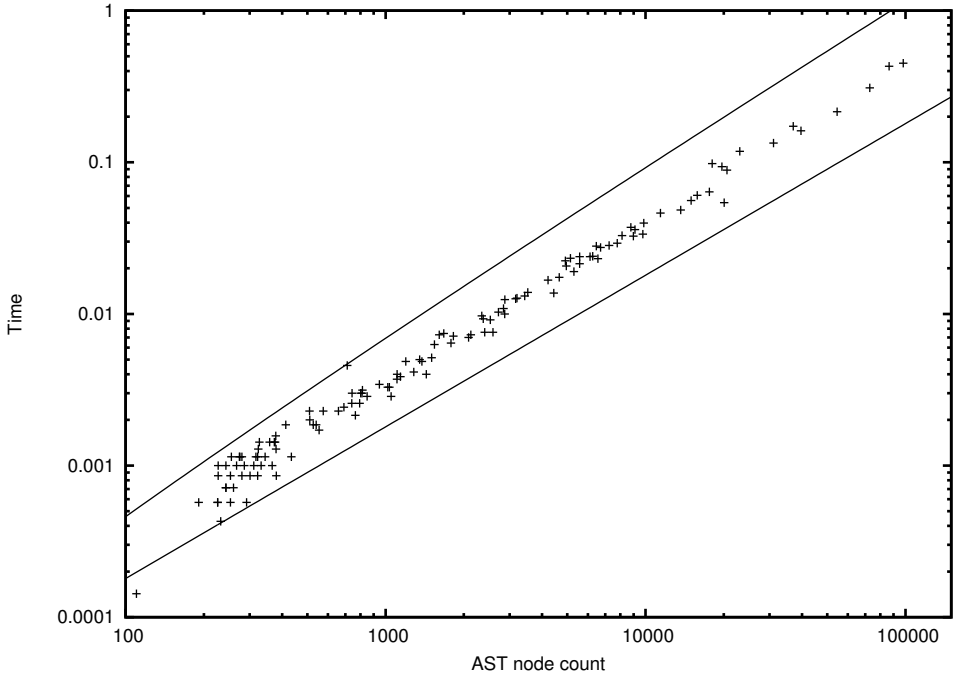


Figure 7.4.: Source node count versus analysis time

8. Related Work

8.1. CFA and CFA-based type recovery

Shivers [1991] uses an extension of 0CFA to perform type recovery. Instead of directly discovering type information about variables, he adds a level of indirection and discovers information about the *quantity* a variable contains. This approach allows information learned about one variable to be shared with its aliases but leads to potential correctness problems if multiple quantities flow to the same variable. Shivers addresses this by introducing a reflow semantics to correct for the problems caused by the indirection around quantities. The analysis presented in this dissertation does not handle quantity information and instead relies on a pass earlier in the compiler that performs copy propagation and aggressive inlining. This keeps the analysis relatively simple while still yielding some of the benefits of his quantities. Since it is based on 0CFA rather than sub-0CFA, Shivers's analysis is more precise though asymptotically more expensive than the analysis in this dissertation.

Serrano [1995] argues that 0CFA is useful in compilers for functional languages by presenting two use cases: an analysis for reducing closure allocation and an analysis for reducing dynamic type tests. Serrano reports that the latter algorithm eliminates 65% of dynamic type tests. This differs from the results in Chapter 7, which show that a 0CFA-based analysis eliminates only 43% of type tests. The difference is likely attributable to

different set of benchmarks being used, different strategies for inserting and counting type checks, and different optimizations leading into the analysis. As it is based on 0CFA, Serrano's analysis takes $O(n^3)$ time in the worst case.

Heintze and McAllester [1997c] describe a linear-time CFA. It is specifically targeted at typed languages and assumes bounds on the sizes of types. In linear time it can list up to a constant number of targets for all call sites, and in quadratic time it can list all targets for all call sites. Mossin [1998] independently developed a similar quadratic analysis for explicitly typed programs based on *higher-order flow graphs*. Whereas these analyses are based on *inclusion*, Henglein's *simple closure analysis* [Henglein, 1992b] computes a cruder approximation based on equality constraints and can be solved in almost linear time via unification. None of these analyses are flow sensitive.

The notion of sub-0CFA defined in Section 2.4 is close to that of Ashley and Dybvig [1998]. They effectively use a more restrictive lattice but provide a general framework through which more general lattices can be constructed. Their analysis achieves a limited form of flow sensitivity when the test of an `if` is a type predicate applied to a variable by creating new bindings for the variable in the *then* and *else* parts of the `if` whose abstract values are restricted by the test. They also describe a more general form of flow sensitivity that tracks variable assignments. It does not gather observational information from nested conditionals, type-restricted primitives, or user-defined functions, and they do not make any claims about its asymptotic behavior.

8.2. Type recovery based on type inference

Soft typing [Cartwright and Fagan, 1991] and more recently, gradual typing [Siek and Taha, 2006] are designed to produce, through type inference, statically well-typed programs from dynamically typed programs by introducing run-time checks or casts. CFA-based type

recovery can be seen as an alternative mechanism for accomplishing a similar effect. While soft typing and gradual type systems might reject some programs, the CFA-based analysis never rejects programs, because type errors are semantically required to cause run-time exceptions.

Henglein [1992a] presents a fast $O(n\alpha(n))$ tagging optimization algorithm for Scheme. Using the terminology of Steenkiste [1991], the goal of the algorithm is to statically eliminate dynamic *tag insertion* and *tag removal* operations. In contrast, the analysis presented in this dissertation seeks to eliminate dynamic *tag checks*. (Values in Chez Scheme are always tagged as the garbage collector relies upon them.) Henglein reports that his algorithm is able to eliminate around 40% of the executed tag insertion operations and around 55% of the executed tag removal operations across six non-numerical operations. Although related, these numbers are not comparable to the number of eliminated tag checking operations reported in Chapter 7. In a companion paper, Henglein [1994] treats the theory of *dynamic typing* in the form of a calculus with explicit type coercions and an equational theory.

The concept of *occurrence typing* developed in the context of Typed Scheme [Tobin-Hochstadt and Felleisen, 2010], is closely related to the present analysis in that different occurrences of the same variable are typed differently depending on the control flow through type-testing predicates. The type system of Typed Scheme expresses types as formulas in a propositional logic that has some similarities to the lattice structure underlying the analysis presented in this dissertation.

8.3. Recent type-recovery applications

Jensen et al. [2009] develop a type analysis for JavaScript. Their analysis is context-sensitive and incorporates both *recency abstraction* and *abstract garbage collection*. They

focus, however, on precision over computational complexity. As a result, their analysis sometimes requires a few minutes to process JavaScript programs of only several hundred lines.

Vardoulakis and Shivers [2010] describe a *summarization*-based CFA with a degree of flow sensitivity. In addition to precise call-return matching, their analysis models precisely the top stack frame of arguments. However, their focus is more on precision than efficiency. The analysis has since been re-targeted to JavaScript in the form of DoctorJS [Mozilla Corporation, 2011].

For type checking dynamically typed programs, Guha et al. [2011] combine a type system and a flow analysis such that the latter boosts the precision of the former. Like the analysis in this dissertation, their flow analysis is flow-sensitive and computes tag sets for each occurrence of a variable. However, it is not interprocedural. Instead it relies on the type system at function boundaries. It has a quadratic worst-case time complexity.

8.4. Other related work

To prove soundness of the analysis, the concretization framework of abstract interpretation [Cousot and Cousot, 1992] can be used. Cousot and Cousot [1979] originally used traces (paths) over a flow graph to prove soundness of classical data-flow analyses. Flow graphs were later generalized to transition systems [Cousot, 1981], and paths were extended to traces thereof.

Wegman and Zadeck [1991] formulated fast constant propagation algorithms for a first-order imperative language. Their *conditional constant propagation* relates to the analysis in this dissertation in that they track reachability and may gain information from a test in a conditional. Wegman and Zadeck list elimination of run-time type checks in a LISP dialect as a possible use case of their approach. Whereas they consider multiple ways to

handle functions, including aliasing of pass-by-reference parameters, they do not consider how to handle first-class functions.

As an illustration of a general *property simulation* algorithm in ESP, Das et al. [2002] instantiate their general framework to a flow-sensitive constant-propagation algorithm. The resulting work-list algorithm is polynomial, however, as it involves invoking a theorem prover at each conditional expression for symbolic evaluation.

9. Future Work

The research presented in this dissertation leaves many areas undeveloped that are worth exploration. This chapter outlines potential future directions for research, which not only improve the precision and performance of the analysis but also use the techniques developed in this research to improve results in other domains.

9.1. User-defined types

Two limitations of the current analysis are that it does not intelligently handle user-defined types such as those generated by the R6RS `define-record-type` [Sperber et al., 2007] and that the internal structure of objects such as pairs or vectors is not tracked.

Adding support for user defined types must be done carefully. If we simply give each user defined type a fresh tag element in $\widehat{Tag}$, then the resulting type lattice would be linear in the number of types in the program. Thus, in a program of size n, which defines $O(n)$ types, the lattice would have a height that is $O(n)$ in size of the program. This linear height lattice voids the $O(n \log n)$ time bound. This situation gets even worse with generative types, which in R6RS allow the generation of an arbitrary number of types at run time.

To resolve this we can use the same trick as was used with procedures. Rather than splitting $\widehat{Val}$ into two parts, $\widehat{Fun}$ and $\widehat{Tag}$, we split it into three parts, $\widehat{Fun}$, $\widehat{Tag}$ and a new part for user-defined types. Then, just as in Section 2.4, we can impose a flattened lattice

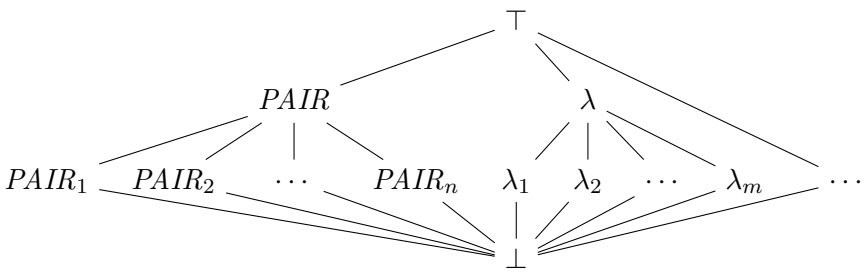


Figure 9.1.: Lattice for rich types

over the user-defined types. However, it might not be a good idea to flatten the lattice to only a single layer as multiple user-defined types may flow to the same expression more often than multiple procedure do. Instead, it may be advisable to use some other lattice that has a bounded height and that takes into account relationships between types such as inheritance or the expression from which generated types come.

9.2. Objects and rich types

Not tracking the contents of objects like pairs, vectors, or user defined types causes the contents to not only be approximated by ⊤ but also to escape. An example is the following expression.

```
(let ([x (cons 2 '())]) (+ 1 (car x)))                    (9.1)
```

Here `(car x)` will always evaluate to the number 2. However, since the analysis currently abstracts all pairs to the single *PAIR* tag, the analysis cannot determine this. We could add a richer type system such as those used in Typed Racket [Tobin-Hochstadt and Felleisen, 2010] but we must be careful to bound the sizes of our types. Otherwise, the analysis may loose its $O(n \log n)$ bound.

As an alternative, we could use the same technique as is already used for functions. Rather than inferring a type for a function, control-flow analysis uses the function itself

as the type. Similarly, we can use the construction location of a pair, vector, or record as the type of an object [Ashley and Dybvig, 1998]. Each construction location then contains information about the values that objects constructed at that location may contain. In Example 9.1 the information would indicate that the pair constructed by the `cons` has a numeric value as its first element and null as its second element.

As with user-defined types this could also cause the lattice of types to be linear in the size of the program and thus break the $O(n \log n)$ bound. The same techniques used there should also solve this issue. However in this case it may be advisable to use a lattice like the one shown in Figure 9.1. Each construction site for a pair would have its own type such as one of $PAIR_1$, $PAIR_2$, and so on, but when more than one pair flows to the same place, the lattice goes up only to the generic $PAIR$ type rather then going to $\top$.

An added complication of adding pairs is that the fields within them are mutable. However, the techniques for mutable variables in Section 6.7 should be sufficient to solve this problem.

9.3. Asymptotic improvements

As currently expressed, the analysis takes $O(n \log n)$ time. The only part of the analysis that is not linear, however, is the use of the skipping functions from the cache to construct the skipping functions for the contexts selected in Section 4.3. An asymptotic improvement to this part of the analysis would improve the asymptotic performance of the entire analysis. While at present there is no obvious method of improving this bound, it is a tempting target for future research given that it is the only part preventing a linear-time analysis.

9.4. Loops and general control flow

While the basic analysis in Chapter 3 is suitable for unstructured programs, the optimizations in Chapter 4 work only on structured programs, i.e., programs which, except for function calls, have planar, acyclic control flow graphs. This is sufficient for Scheme as Scheme's loops are expressed in terms of function calls. However, many languages have loops which introduce cycles or jumps that introduce non-planarity. Further, a direct handling of loops and jumps is likely to yield more precise results than encoding them as function calls.

In order to generalize the analysis to handle these cases, the selection and computation of contextual skipping functions needs to be generalized. In the case of loops, skipping functions need to handle when variables flow back into themselves, and preliminary work indicates the feasibility of the approach. However, it is unclear whether the algorithm can be generalized to non-planar graphs.

9.5. Static single assignment form

Static single-assignment form (SSA) is a well known and common method of restructuring a program [Rosen et al., 1988, Alpern et al., 1988]. However, converting a program to SSA form can increase the size of a program up to $O(n^2)$ where n is the size of the original program [Cytron et al., 1991]. This size increase is due to the introduction of potentially quadratically many ϕ-nodes into the program. These ϕ-nodes serve a similar purpose to the contexts selected by the context selection algorithm in Section 4.3, and the context selection algorithm can be viewed as an alternative to SSA that offers the advantage of taking only linear space in the worst case and thus does not asymptotically increase the size of the program. Thus, the representation used here may be quite useful for implementing other typical compiler optimizations and algorithms and would result in

a lower asymptotic bound for those operations. In the general case, this works only if loops and non-structured graphs can be handled. Thus, the handling of loops and general graphs discussed in Section 9.4 has implications beyond the control-flow analysis described in this dissertation. If the techniques in Chapter 4 do not generalize to cyclic and non-planar graphs, then SSA is still useful for programs that exhibit those structures. However, if they do generalize, then they are a viable alternative to SSA.

9.6. Sub-kCFA

Sub-0CFA is essentially the same as 0CFA except that a widening operator is used to ensure better asymptotic bounds. Variants of CFA that store more contextual information about a function call, such as 1CFA, offer better precision but at the cost of performance. For example, 1CFA is EXPTIME complete. In order to gain the advantage of precision from more contextual information without paying the performance costs, it may be possible to apply the same widening trick that sub-0CFA uses. It is likely that the widening operator needs to work over contexts in addition to types, but the basics of the theory would be similar. In this way the techniques developed here to achieve an $O(n \log n)$ asymptotic bound could be ported to other analyses.

9.7. Generalized skipping functions

The analysis presented here tracks only the relationship between the truth value returned by an expression and the types of variables. This concept can be generalized. Instead of storing two environments with an expression, one for when the expression is true and one for when the expression is false, the analysis can store one environment per type that the expression returns. Or more naturally, it can store a function that maps from a return type to the environment that must hold when that type is returned from the expression.

To match this generalization, the output types of function arguments must also change to have a value for each variable for each type that the function can return. Care must be taken with this construction to avoid increasing the lattice height, but otherwise it is fairly straightforward.

This generalization has several advantages. First, the rules for application become significantly simplified. As they are currently, each sub-expression of an application has to be handled differently depending on whether it is a variable or not. For example, this is done in the *ARG* function. By generalizing, these rules can be more uniform and do not need special cases for variables.

Second, as a consequence of eliminating the special case for variables, a more precise analysis becomes possible. For example, without the generalization, the expression `(pair? (begin` e_1 `x))` would not provide any information about `x`, but with the generalization, it would gather as much information as `(begin` e_1 `(pair? x))`. Another common case is `(not (pair? x))`. Since `not` is a function instead of a macro in Scheme, without the generalization, this expression gathers no information about `x`, but with the generalization, it gathers just as much information as `(if (pair? x) #f #t)`.

Finally, this generalization allows the implementation of an efficient *backwards abstract interpretation* as described in Section 9.8.

To implement this, care must be taken to ensure the lattice over the function mapping types to environments, has a bounded-height lattice. For example, if each procedure type had its own environment, then it could break the linear-log time bound. Even if the lattice is arranged to prevent breaking the asymptotic time bound, we must be careful that the constants in a practical implementation do not grow too large. Doing this efficiently might be challenging.

9.8. Backwards abstract interpretation

The connection between abstract interpretation [Cousot and Cousot, 1977, 1979] and control-flow analysis is well known [Midtgaard and Jensen, 2008], but what Cousot and Cousot [1999] refer to as backwards abstract interpretation has a particularly close connections to the control-flow analysis defined here. With standard, forwards abstract interpretation, the potential values of sub-expressions are used to discover the potential values of enclosing expressions. For example, in `(if (pair? x)` e_1 e_2`)`, the potential values of `x` are used to determine the potential values of `(pair? x)`, which in turn determine if e_1 and e_2 are reachable. However, backwards abstract interpretation starts with potential values of enclosing expressions and deduces potential values of sub-expressions. For example the enclosing `if` will pass control to e_1 only if `(pair? x)` returns true. Also `(pair? x)` returns true only if `x` is a pair. This then allows the analysis to conclude that `x` must be a pair if e_1 is reachable.

The analysis presented in this dissertation achieves the same effect. However, combining both forwards and backwards abstract reasoning usually requires alternating between the two as the system uses information from one to incrementally improve the other. The analysis presented in Chapter 3 avoids this alternation. Instead of deriving backwards from the possible values of an expression to the possible values of the sub-expressions, each expression directly produces information about sub-expressions given the possible values of the expression. For example, `(pair? x)` has that `x` is a pair in the true case and that `x` is not a pair in the false case.

This trick is a simple one. In fact it is also used with attribute grammars [Knuth, 1968] where inherited properties can be encoded as higher order synthetic properties. In the analysis, however, it not only avoids needing to alternate between backward and forward modes, but as shown in Chapter 4 it also allows an efficient implementation of the analysis.

It is worth further research to study how these results translate from control-flow analysis to abstract interpretation.

10. Conclusion

This dissertation describes a flow-sensitive type-recovery algorithm based on sub-0CFA that runs in linear-log time. It justifies, on average, the removal of about 60% of run-time type checks in a standard set of benchmarks for the dynamically typed language Scheme. It handles, on average 100,000 lines of code in less then a second.

Research in this area often focuses on developing higher precision analyses at the cost of worst-case performance. Often this is excused by saying that the algorithm, while having bad worst-case performance, performs well "in practice".

There is some justification to this as higher precision analyses often do "in practice" perform better because a more precise approximation means less time analyzing code paths that cannot actually occur [Might and Shivers, 2006].

However, we need to ask ourselves whether we want an analysis that is guaranteed to perform well but in a few cases is less precise, or an analysis that offers a slightly better precision but does not guarantee performance. Both sorts of analyses may "in practice" be both precise and perform well. It is a question of whether the analysis should sacrifice performance or precision for worst-case inputs. From a compiler optimization perspective, "lots of little bullets" is often better then one big bullet [Peyton Jones et al., 1996]. One big bullet can miss, but lots of little bullets are unlikely to all miss.

I conjecture that many existing analyses can be modified to add a time bound guarantee if we are willing to accept some loss of precision on worst-case inputs. Research into

higher precision analyses should include studying how to limit the computation time of the analysis. In some cases, like the transition from 0CFA to sub-0CFA, adding such a time bound is fairly direct and easy. There is little excuse for not showing how to bound the time in these cases. In others, like the flow sensitivity implemented in this dissertation, it may be much harder, but the results are likely to advance the state of the art.

The implementation conservatively handles the unspecified evaluation order of arguments and bindings. Making evaluation-order decisions earlier in the compiler would allow the analysis to produce more precise information, particularly if the decisions were influenced by the needs of the analysis. Experiments show that the typical benefit is likely to be minimal, but the benefit in some cases would be substantial.

Employing an extended lattice that differentiates pairs and vectors based on their allocation sites as the analysis already does for functions, should also lead to more precise information. In a statically typed variant of the analysis, the lattice can also be refined to differentiate functions with different static types. Even in a dynamically typed language, functions can be grouped by arity.

Another avenue for further investigation is to supplement the current techniques with an efficient *must-alias analysis*, such that for two aliased variables `x` and `y`, information learned about `x` is reflected in `y`. The higher-order must-alias analysis by Jagannathan et al. [1998] is a natural starting point for such an investigation.

Finally, I conjecture that the same techniques used to extend sub-0CFA with flow sensitivity can be applied more generally to kCFA with the addition of a single logarithmic factor to the asymptotic cost.

A. Source Code Overview

The complete source code implementing the analysis described in this dissertation is included in appendix B. It is also available in digital form from the author upon request and at the following URL:

`http://www.cs.indiana.edu/~adamsmd/phd/`

The remainder of appendix gives a short overview of the code to help orient the reader who wants to understand the code. Indexes to file, function, macro and record names are included at the end of this dissertation.

A.1. Using the code

The code runs and is tested under Chez Scheme 8.3. It should also run on other versions and other Scheme systems that support R6RS [Sperber et al., 2007]. In a few cases, Chez Scheme specific forms may need to be ported if using another Scheme system.

The simplest way to start experimenting with the code is to load and import `driver.ss` with `(import (driver))` at the REPL. Once loaded, it defines the `run` function which takes an S-expression as input and runs the analysis on the input. For example, the input

```
(run '(let ([x (read)]) (car x) (if (pair? x) 1 2)))
```

returns the following output.

```
(letrec ([x.1 (read)])
  (begin (car x.1) (begin (pair? x.1) '1)))
```

The `tests.ss` file also contains a few basic tests of the analysis. After importing the `(tests)` library, calling `(run-tests)` will run all tests and print a report of which ones produced their expected output.

The `run` function uses a simplistic parser defined in `parse-scheme.ss`. For more deep integration with a compiler, the function `run-type-recovery` takes an already parsed expression in the form of a `Trex` record (see Section A.3.1). This function is the entry point to the algorithm used by the driver. The `run` function in `driver.ss` is merely a wrapper around `run-type-recovery` with a parser from S-expressions to `Trex` records.

A.2. Algorithm

The main algorithm starts at `run-type-recovery` in `type-recovery.ss`. In addition to construction the flow graph (see Section A.2.2), it also does preprocessing (see Section A.2.1) and post-processing (see Section A.2.3) to prepare the input for the construction of the flow graph and utilize the flow graph results, respectively.

A.2.1. Preprocessing

Before the flow graphs are constructed a bit of preprocessing is done. First, `transform-let` in `transform-let.ss` replaces any occurance of `((lambda (`*var*`...) `*body*`)` *arg*`...)` with `(letrec ([`*var* *arg*`]...)` *body*`)` in order to force the order of evaluation between *arg* and *body* as discussed in Section 6.3. The `letrec` form is used here since the analysis assumes all bindings in the input have unique variable names. In that case, `letrec` is a correct implementation of `let` except for the differences in handling continuations. By having only the `letrec` form, the code becomes simpler and has one fewer case to handle. This makes implementing this research prototype easier, but before production use, the implementation needs to be changed to distinguish between the two.

Next, the algorithm places a wrapper around the expression to mark that the result of the expression should be marked as escaped (see Section 2.3) with `escape-top-level-lambda`. Then the `Trex` record is converted to an `AST` record via the `Trex->AST` function defined in `ast.ss` (see Section A.3.2). During the process of conversion, a Myers stack is constructed starting at each AST node and going up to the root of the `AST` record.

Finally, `make-AST-bubbles` from `ast-bubbles.ss` preprocesses the `AST` record to determine what contexts to skip based on the rules described in Section 4.3. In the process, the locations where each identifier is referenced are collected using `tree-locations.ss`. These locations are stored in `ident-asts` and `ident-locations`. The `ident-locations` help compute the "bubble" calculations efficiently. The `ident-asts` are used elsewhere in the algorithm.

A.2.2. Flow-graph construction

Flow-graph construction is the heart of the algorithm and is done in three passes. In the first pass, the environmental flow-graph nodes are constructed as unforced, lazy flow-graph nodes. In the second, the value flow-graph nodes are constructed. Finally, the environmental flow-graph nodes are forced. See Section A.3.3 for an explanation of flow graphs and lazy nodes. While both sorts of flow graph nodes could be constructed in a single pass, constructing them separately simplifies the code.

A.2.2.1. Environmental flow-graph nodes

The environmental flow graph nodes are those that deal with computing the environmental flow functions described in Section 4.1. They are computed by `make-env-flow-graph` in `env-flow-graphs.ss`. The flow of environments entering an expression reduces to simple reachability, so only the environmental flow of environments exiting an expression is computed. These environmental flow functions are stored in the generalization of Myers

stacks described in Section 4.4 and defined in `composition-stacks.ss`. These generalized Myers stacks extend the standard Myers stacks defined in `myers-stacks.ss` store not only the nodes for single steps up the AST as defined in Figure 4.2, but also nodes for the compositions of those steps. Later, when computing the value flow-graph nodes, these are used to construct nodes for moving variable value information from one location to another in the program.

A.2.2.2. Value flow-graph nodes

Computing the flow-graph nodes for everything other than the environmental flow is the responsibility of `make-value-flow-graph` in `value-flow-graphs.ss`. This function recurs over the AST and computes the necessary parts of the graph.

A complication is that, while the environmental flow nodes already exist, the function still must construct nodes to flow individual variables through the program. This is done by storing a record in each identifier that describes the last reference from which values should flow. These are stored in the `ident-preceding-ref` field as either an `up-ref` or `down-ref` record depending on whether the last reference was flowing out of or into an expression respectively. Then the `move-var-to-here` function, which is called at the end of each recursive call for each variable identified by `make-AST-bubbles`, uses the nodes for environmental flow to compute the result of flowing such variables through the program.

This gets further complicated with the right-hand sides of `letrec` or the arguments of function calls. These have to be computed independently because the analysis does not assume a particular order of evaluation. A similar problem occurs between the *then* and *else* branches of `if` with the primary differences being the values need to be unioned rather than intersected. The solution involves careful manipulation of the `indent-previous-ref` fields. The `parallel*`, `parallel-loop` and `parallel-intersected` macros provide abstractions for this.

The value flow calculations for each of the expression forms generally follow the obvious implementation based on the rules in Figure 3.5. Nevertheless, the code for function application is quite complicated. Not only does it take care of the CALL_{in}, CALL_{fun}, CALL_{mid}, and CALL_{out} rules, which are the most complicated rules in Figure 3.5, but the application form in Scheme is much more complicated than the one presented in Figure 3.1. For example, we have indeterminate order of evaluation, `case-lambda`, and variable argument forms. The code for functions and function calls is complicated relative to the rest of the code. While desirable, it is not yet clear how to more simply express this part of the code.

When constructing the value flow graph, the nodes for references to primitives are calculated based on a table in `prims.ss`.

A.2.3. Post-processing

Once both the environmental and value flow-graph nodes are constructed, the algorithm is almost complete. All that remains is to step the flow graph with `flow-graph-step!` until the graph stabilizes and `flow-graph-stable?` returns true. Then `AST->Trex` from `ast-unparse.ss` converts the `AST` record back to a `Trex` record. In the process it applies optimizations based on the information learned about values and variables.

A.3. Data types

A.3.1. Trex records

The `tprogram.ss` file defines the type of `Trex` records using the `define-datatype` form (see Section A.4.1). They are used in the external interface to the analysis. Internally `AST` records are used.

A.3.2. Abstract syntax trees

The `ast.ss` file defines the type of abstract syntax trees used by the analysis. It defines them via the `define-datatype` form (see Section A.4.1). `AST` node types are as follows:

`Ref` Variable references

`Quote` Constants

`Primref` References to built-in primitive functions

`Set` Variable assignments

`App` Function applications

`If` Conditionals

`CaseLambda` Lambda abstractions of both `case-lambda` and `lambda` forms

`Seq` Sequential execution

`LetRec` Recursive let binding, i.e., `letrec`.

In addition to the data specific to each node type, e.g., the identifier of a `set!`, every node includes the following meta-data:

src Meta-data such as source information that is passed through the analysis unchanged.

output-node The flow-graph node corresponding to the output of this AST object.

contained-nodes A list of flow-graph nodes constructed for this AST object. Used for debugging.

context A Myers stack containing the path from the AST object back to the root of the tree as described in Section 4.4.

graph-context A Myers stack containing the path from the AST object back to the root of the tree that is annotated with environmental flow functions as described in Section 4.4.

graph-context-thunk A thunk used to delay part of the construction of the `graph-context`.

bubble-vars The list of identifiers that that bubble algorithm annotated this node with. These identifiers indicate which contexts to skip for each var as described in Section 4.3.

tree-start-location Used internally by `make-AST-bubbles`.

tree-end-location Used internally by `make-AST-bubbles`.

A.3.3. Flow graphs

A general framework for flow graphs is defined in `flow-graphs.ss`. A flow graph is created with `make-flow-graph`. Then after the appropriate flow-graph nodes are added to the graph, the `flow-graph-step!` function is used to make the flow graph take one step and update a single node. Whether there are more nodes that need updating is determined by the `flow-graph-stable?` function. The `flow-graph-print` function is for debugging purposes and outputs GraphVis `dot` files [Gansner and North, 2000] for visualizing the flow graph.

Each flow graph node has both an output value accessible with `flow-graph-node-output` and arbitrary meta-data accessible with `flow-graph-node-info`. Every flow graph node also takes an `eqv?` predicate at construction time that takes two output values from the flow graph node and returns true if and only if they are equal. This is used to determine whether the output of a node has changed and nodes that depend on the node need to be put on the work list. In addition, when a flow graph node is first constructed, the `eqv?`

function is called with the initial output of the node passed as both arguments. This allows the detection of more type errors at node creation time, rather than later when the flow graph is being stepped.

There are four kinds of flow graph nodes that can be constructed:

- A *function node* is constructed with `make-flow-graph-function-node` and is the most common type of node. It takes a function and a list of nodes that the newly created node depends upon. Whenever one of the nodes that are depended upon is updated, the function is applied to the output values of the nodes depended upon. The result of the function application then becomes the new output value of the node.
- A *lazy node* is constructed with `make-flow-graph-lazy-node` and is like a function node except that the nodes that it depends upon are specified at a later time with `flow-graph-lazy-node-force!`. This allows the construction of flow graphs that contain cycles. All lazy nodes should be forced before calling `flow-graph-step!`.
- A *mutable node* is constructed with `make-flow-graph-mutable-node`. This node has no input and its output is manually modified with `flow-graph-mutable-node-set!`. Care must be taken with this node type to ensure that its output value obeys the rules for flow-graph nodes.
- A *join node* is constructed with `make-flow-graph-join-node` and is a higher-order node. It takes as argument, A, a flow graph node that it depends upon. It expects the output of A to be another flow graph node, A^*. The output value of A^* upon becomes the output of the join node. This construction, though unusual, allows the analysis to analyze function calls, and account for dependencies that are not known when the flow graph is constructed.

The performance of the flow-graph framework defined can likely be improved by eliminating the large number of higher order function calls and indirects used within it.

A.3.4. AST flow graphs

Since most of the flow graphs are attached to AST nodes, the `ast-flow-graphs.ss` file provides several helpers to simplify the use of flow graphs with AST nodes.

A.3.5. Identifiers

Identifiers are represented with the `ident` record type defined in `idents.ss`. Each identifier contains the following fields:

`name` An object that uniquely identifies the `ident`.

`symbol-name` A symbol used for pretty-printing.

`assigned` A boolean that is true if and only if the `ident` is mutated.

`number` A unique integer used for sorting `ident` values.

`asts` A list of `Ref` and `Set` records that reference this `ident`.

`locations` Used internally by `make-AST-bubbles`.

`preceding-ref` Used at graph construction time to store the last reference to the `ident`. It is always either an `up-ref` or a `down-ref` record.

`binding-node` The output node of the value to which the `ident` is bound. For example, if bound by a `letrec`, it is the output node of the binding's right-hand side.

A.3.6. Types

Types are represented by the `type` record in `types.ss`, which contains both a `proc` field for the procedure part of the type and a `non-proc` field for the non-procedure part of the type. The `non-proc` field is a bit mask using bits defined with the `define-mask-bits`

form. The `proc` field contains either `bot-proc-type-bits` representing the $\bot$ element of the procedure lattice, `top-proc-type-bits` representing the $\top$ elements of the procedure lattice, or an object created with `one-or-more-proc-type-bits` representing an element of the procedure lattice containing one or more procedures.

All of these type forms have intersection, union, equality, and lattice comparison (i. e., $\sqsubseteq$) defined for them. In addition many constants and helper functions for manipulating or converting these types are defined.

Finally, since a type for a true case and a false case are so often manipulated together, `bitype.ss` defines a representation of two types bundled together.

A.3.7. Environmental flow functions

The `env-flow-functions.ss` file defines environmental flow functions of the sort specified in Section 4.1. The zero, identity, composition, and application to `bitype` records are all defined. For efficiency reasons, environmental flow functions are represented as bit masks.

A.4. Auxiliary code

Some parts of the source code are not part of the core algorithm but are worth mentioning even though they are merely utility code.

A.4.1. datatype.ss

This file defines the `define-datatype` form. This form defines record types that model an algebraic data type (ADT) and is used to define both `AST` and `Trex` records. Its general form is

`(define-datatype (`*adt-name prop*`...) (`*constructor-name field*`...)...)`

where a *prop* is either `(`*prop-name default*`)` or `(`*prop-name default type*`)`, and a *field* is either a *field-name* or `(`*field-name type*`)`.

The *adt-name* defines the name of the ADT. It is used to define a type predicate which will be named *record-name*`?` that returns true if and only if its argument is of type *adt-name*. Its also used to define a case construct for the ADT.

Each *prop* defines a property that every ADT object has. The *prop-name* is used to define accessors and mutators named *adt-name-prop-name* and *adt-name-prop-name*`-set!` respectively. The *default* is the value that the property has when the ADT object is constructed and is required as constructors do not take properties as arguments. The *type*, if present, is a predicate that returns true only for values of the correct type for the property and is used by the property mutators to check the type to which the property is set.

Each (*constructor-name field*...) clause defines a new constructor for the ADT. The *constructor-name* is used to define the function for constructing a value of that type as well as a predicate to test for values of that type. The name of the constructor is *constructor-name* and the name of the predicate is *constructor-name*`?`. Each *field* specifies an argument to the constructor. If present, the *type* is a predicate that returns true only for values of the correct type for the field and is used by the constructor to check the type of a field value passed to it.

Finally, for each ADT, a case form is defined with the following syntax.

(*adt-name*`-case` *expr* `[(`*constructor-name field-binding*...`)` *body*`]`...
`[else` *else-body*`])`

The *expr*, *body* and *else-body* sub-forms are expressions. A *constructor-name* is the name of one of the constructors and a *field-binding* is an identifier. The `else` clause may be omitted. if is not present, then every *constructor-name* in the ADT must be present in the case form, but order does not matter. If a *constructor-name* is missing and there is no `else` clause, compile-time exception is raised.

The *expr* must return an ADT object. If the ADT object is constructed from a listed *constructor-name*, each *field-binding* is bound to the corresponding field originally passed to the constructor, and the corresponding *body* is executed. If no *constructor-name* matches, then the *else-body* is executed.

Finally, this file also defines `list-of`, `vector-of`, and `improper-list-of`, each of which takes a predicate and returns a predicate that returns true if and only if its argument is a list, vector, or improper list that contains elements for which the argument predicate returns true.

A.4.2. records-io.ss

This file exports the `record-writer` and `record-reader` forms defined in Chez Scheme [Dybvig, 2010] which tell the pretty printer how to print certain record types. The purpose this library is to making porting to other systems easier. In order to port to a system that does not have `record-writer` and `record-reader` forms, this library needs to be replaced with one that defines them as no-ops. However, without the overrides that `record-writer` and `record-reader` provide, care must be taken when when printing `AST` records as they contain both cycles and large amounts of meta-data.

A.4.3. define-ignored.ss

This file defines the `define-ignored` macro. This macro expands expressions that are of the form `(define-ignored` *expr*`)` into `(define` *ignored expr*`)`, where *ignored* is a fresh variable name. This form is used in a number of places in the code when the side effects of *expr* are needed without changing from definition context to expression context [see Sperber et al., 2007, section 11.3].

A.4.4. iota.ss

This file defines an `iota` function that takes a single non-negative integer as argument and returns a list of integers from zero up to but not including the argument.

A.4.5. safe-forms.ss

This file defines safer, alternate versions of `if`, `cond`, and `case`. The `if` does not allow one-armed forms. The `cond` raises an error if no cases of the `cond` succeed. The `case` raises an error if no cases match. The `case` also allows bare symbols in match clauses. These are all modified from the forms in Appendix B of Sperber et al. [2007].

A.4.6. syntax-helpers.ss

This file defines several functions that make it easier to write `syntax-case` macros [Dybvig et al., 1993]. The `syntax-ellipses?` function returns true if and only if its argument is the syntax object for ellepsis. The `syntax-indexes` function maps a syntax-object like `(a b c)` to a syntax object with each element annotated with its index in the object such as `((0 . a) (1 . b) (2 . c))`. The other functions (`syntax-length`, `syntax-car`, `syntax-cdr`, `syntax-map`, `syntax-for-all`, and `syntax-append`) are similar to their list counterparts (`length`, `car`, `cdr`, `map`, `for-all`, and `append`) but operate on syntax objects rather than lists.

A.5. Third-party code

A.5.1. srfi-39.ss

This file implements the subset of "SRFI 39: Parameter objects" [Feeley, 2003], that is needed by the code. If your Scheme implementation has a native form of parameters, you can substitute that for this library.

A.5.2. match.ss

This file implements an S-expression based pattern matcher. See the following URL for details on the syntax of the matcher.

`http://www.cs.indiana.edu/chezscheme/match/`

B. Source Code

This appendix includes a source listing of the code that implements the analysis described in this dissertation. This code is provided for informative purposes only and should be considered research quality software. It includes both instrumentation code and code that needs performance tuning. It is not production ready software.

B.1. Software license

The following copyright and license applies to all code in this appendix with the exception of `match.ss`. The copyright and license for `match.ss` are contained in `match.ss` itself.

B.2. Algorithm

Listing B.1: driver.ss

```
(library (driver)
  (export run)
  (import (rnrs) (type-recovery) (parse-scheme))

(define (run expr) (unparse-scheme (run-type-recovery (parse-scheme expr))))

)
```

Listing B.2: type-recovery.ss

```
(library (type-recovery)
  (export run-type-recovery)
  (import (rnrs) (ast) (ast-flow-graphs)
          (ast-bubbles) (ast-unparse) (env-flow-graphs) (value-flow-graphs)
          (flow-graphs) (transform-let) (types) (prefix (tprogram) Trex:))

(define (escape-top-level-lambda x)
   (Trex:App #f (Trex:Primref #f #f 'escape) (list x)))

;; Trex -> Trex
(define (run-type-recovery trex)
  (let ([flow-graph (make-flow-graph)])
    (with-AST-flow-graph flow-graph
      (let ([ast (Trex->AST (escape-top-level-lambda (transform-let trex)))])
        (make-AST-bubbles ast)

        ;; create the flow graph nodes
        (make-env-flow-graph ast)
        (make-value-flow-graph ast)
        (env-flow-graph-force! ast)

        ;; run the flow graph
        (let loop ()
          (unless (flow-graph-stable? flow-graph)
            (flow-graph-step! flow-graph)
            (loop)))

        (AST->Trex ast)))))

)
```

Listing B.3: parse-scheme.ss

```
(library (parse-scheme)
  (export parse-scheme parse-unsafe-scheme unparse-scheme)
  (import (rnrs) (tprogram) (match) (idents))
```

```
;; This module parses s-expressions into Trex records.  It is for demonstration
;; purposes only and should not be considered a robust scheme parser.

;; unique-name produces a unique name derived the input name by
;; adding a unique suffix of the form .<digit>+.
(define unique-name
  (let ([count 0])
    (lambda (sym)
      (set! count (+ count 1))
      (string->symbol
       (string-append (symbol->string sym) "." (number->string count))))))

;; improper-map -- helper function for getting a list of unique variables
;;                 when we get imporper formals lists.
(define (improper-map f ls . more)
  (if (null? more)
      (let map1 ((ls ls))
        (cond
         [(null? ls) '()]
         [(pair? ls) (cons (f (car ls)) (map1 (cdr ls)))]
         [else (f ls)]))
      (let map-more ((ls ls) (more more))
        (cond
         [(null? ls) '()]
         [(pair? ls) (cons (apply f (car ls) (map car more))
                           (map-more (cdr ls) (map cdr more)))]
         [else (apply f ls more)]))))

(define (improper-memq s ls)
  (cond
   [(null? ls) #f]
   [(and (pair? ls) (eq? s (car ls))) (cdr ls)]
   [(pair? ls) (improper-memq s (cdr ls))]
   [else (and (eq? s ls) '())]))

(define who 'parse-scheme)

;; parse-scheme accepts a single value, verifies that the value
;; is a valid program in the source language, replaces bound variables
;; with unique variables, and simplifies several kinds of expressions.
;;
;; The grammar changes from Assignment 14 in that it allows unquoted
;; fixnum, boolean, and empty-list constants; variables are arbitrary
;; symbols; lambda, let, and letrec bodies are implicit begin expressions;
;; one-armed if expressions are allowed; unquoted fixnums and booleans
;; are allowed; quoted vector/list structure is allowed; "and" expressions
;; are allowed; and "or" expressions are allowed.  In addition, "not" is
;; included in the set of primitives.
;;
;;  Program --> <Expr>
;;  Expr    --> <prim>
;;            |  <constant>
;;            |  <var>
;;            |  (quote <datum>)
;;            |  (if <Expr> <Expr>)
;;            |  (if <Expr> <Expr> <Expr>)
;;            |  (and <Expr>*)
;;            |  (or <Expr>*)
;;            |  (begin <Expr>* <Expr>)
```

```
;;            |  (lambda (<var>*) <Expr>+)
;;            |  (let ([<var> <Expr>]*) <Expr>+)
;;            |  (letrec ([<var> <Expr>]*) <Expr>+)
;;            |  (set! <var> <Expr>)
;;            |  (<Expr> <Expr>*)
;;
;; Where constant is #t, #f, (), a symbol or an exact integer
;;       var is an arbitrary symbol
;;       datum is a constant, pair of datums, or vector of datums
;;       primitives are void, error, read (zero arguments); car, cdr,
;;         vector-length, make-vector, not, boolean?, integer?, bignum?,
;;         fixnum?, null?, pair?, procedure?, symbol?, vector?
;;         (one argument); *, +, -, /, <, <=, =, >=, >, fx*, fx+, fx-,
;;         fx/, fx<, fx<=, fx=, fx>=, fx>, cons, vector-ref, eq?,
;;         set-car!, set-cdr! (two arguments); and vector-set!
;;         (three arguments).
;;
;; A var may not be bound multiple times by the same lambda, let, or
;; letrec expression, but a var may be bound by multiple lambda, let,
;; and letrec expressions, with its scope determined via the usual
;; Scheme rules.  No names are reserved, so a keyword or primitive name
;; loses its special meaning within a the scope of a lambda, let, or
;; letrec binding binding of the name.

(define (make-seq expr*)
  (define (strip-begin-seq expr*)
    (match `(begin ,@expr*)
      [(begin ,[expr*] ...) (apply append expr*)]
      [(seq ,[expr*] ...) (apply append expr*)]
      [,expr (list expr)]))
  (define (make-seq expr*)
    (match expr*
      [(,x) x]
      [(,x* ... ,x) (Seq #f (make-seq x*) x)]))
  (make-seq (strip-begin-seq expr*)))

;;; expr -> var
;;;         (quote src datum)
;;;         (primref src safe name)
;;;         (set! src var expr)
;;;         (app src expr expr*)
;;;         (if src test then else)
;;;         (case-lambda src clause*)
;;;         (seq src expr expr)
;;;         (letrec src ([var expr]) expr)
;;;         (type/expr src expr ty key stuff ...)  ; unconditionally evaluate expr,

(define (verify-var-list x*)
  (let loop ([x* x*])
    (cond
     [(null? x*) #f]
     [(pair? x*)
      (let ([x (car x*)] [x* (cdr x*)])
        (unless (or (symbol? x) (ident? x))
          (assertion-violation who "invalid␣list␣found" x))
        (when (improper-memq x x*)
          (assertion-violation who "non-unique␣var␣list␣found" x))
        (loop x*))]
     [else (unless (or (symbol? x*) (ident? x*))
```

```
            (assertion-violation who "invalid␣var␣list␣found" x*))])))

(define (constant? x)
  (or (memq x '(#t #f ())) (and (integer? x) (exact? x)) (string? x) (char? x)))

(define (datum? x)
  (or (constant? x)
      (symbol? x)
      (if (pair? x)
          (and (datum? (car x)) (datum? (cdr x)))
          (and (vector? x) (for-all datum? (vector->list x))))))

(define (uvar-maker sym) (unique-name sym))
(define (make-uvar-ident symbol)
  (let ([x (uvar-maker symbol)]) (make-ident x x #f)))
(define (wrap-var v) (if (ident? v) v (make-uvar-ident v)))
(define (unique-var) (make-uvar-ident 't))

(define empty-env '())
(define (extend env id x) (if (ident? id) env (cons (cons id x) env)))
(define (lookup id env) (cond [(assq id env) => cdr] [else id]))

(define (extend-variables env var* uvar*)
  (cond
   [(null? var*) env]
   [(pair? var*) (extend-variables
                  (extend env (car var*) (car uvar*))
                  (cdr var*) (cdr uvar*))]
   [else (extend env var* uvar*)]))

(define prim-list
  '(+ - * / <= < = >= > fx+ fx- fx* fx/ fx<= fx< fx= fx>= fx> $fxu<
    boolean? car cdr cons eq? integer? bignum? fixnum? symbol? make-vector
    null? pair? fxzero? procedure? set-car! set-cdr!
    vector? vector-length vector-ref vector-set!
    void error display read memq for-all))

(define (Expr env safe)
  (lambda (x)
    (define Application
      (lambda (env proc args) (App #f proc (map (Expr env safe) args))))

    (match x
      [,expr (guard (Trex? expr)) expr] ;; HACK
      [,expr (guard (ident? expr)) (Ref #f expr)] ;; HACK
      ;; bound identifiers
      [,id (guard (and (symbol? id) (ident? (lookup id env))))
           (Ref #f (lookup id env))]
      [(,id ,arg* ...) (guard (and (symbol? id)) (ident? (lookup id env)))
       (Application env (Ref #f (lookup id env)) arg*)]
      [,id (guard (and (symbol? id) (memq id prim-list)))
       (Primref #f safe id)]

      ;; special forms
      [,k (guard (constant? k)) (Quote #f k)]

      [(quote ,datum)
       (assert (datum? datum))
       (Quote #f datum)]
```

```
      [(if ,[t] ,[c] ,[a])
       (If #f t c a)]
      [(if ,[t] ,[c])
       (If #f t c (App #f (Primref #f #t 'void) (list)))]
      [(not ,[x])
       (If #f x (Quote #f #f) (Quote #f #t))]

      [(and ,[x*] ...)
       (if (null? x*)
           (Quote #f #t)
           (fold-left (lambda (a b) (If #f a b (Quote #f #f))) (car x*) (cdr x*)))]
      [(or ,[x*] ...)
       (if (null? x*)
           (Quote #f #f)
           (fold-left
            (lambda (a b)
              (let ([t (unique-var)])
                (let ([t-ref (Ref #f t)])
                  (App
                   #f (CaseLambda
                       #f (list (CaseLambdaClause (list t) (If #f t-ref t-ref b))))
                   `(,a)))))
            (car x*) (cdr x*)))]

      [(begin ,[x] ,[x*] ...)
       (make-seq `(,x ,x* ...))]

      [(lambda ,fml* ,x ,x* ...)
       (verify-var-list fml*)
       (let ([uvar* (improper-map wrap-var fml*)])
         (let ([env (extend-variables env fml* uvar*)])
           (CaseLambda
            #f (list (CaseLambdaClause
                      uvar* (make-seq (map (Expr env safe) `(,x ,x* ...)))))))))]

      [(case-lambda [,fml** ,x* ,x** ...] ...)
       (for-each (lambda (fml*) (verify-var-list fml*)) fml**)
       (let ([uvar** (map (lambda (ls) (improper-map wrap-var ls)) fml**)])
         (let ([env* (map (lambda (fml* uvar*) (extend-variables env fml* uvar*))
                          fml** uvar**)])
           (let ([body* (map (lambda (x x* env)
                               (make-seq (map (Expr env safe) `(,x ,x* ...))))
                             x* x** env*)])
             (CaseLambda #f (map CaseLambdaClause uvar** body*)))))]

      [(let ([,lhs* ,[rhs*]] ...) ,x ,x* ...)
       (verify-var-list lhs*)
       (let ([uvar* (map wrap-var lhs*)])
         (let ([env (extend-variables env lhs* uvar*)])
           (App
            #f (CaseLambda
                #f (list (CaseLambdaClause
                          uvar* (make-seq (map (Expr env safe) `(,x ,x* ...)))))))
            rhs*)))]

       [(letrec ([,lhs* ,rhs*] ...) ,x ,x* ...)
        (verify-var-list lhs*)
        (let ([uvar* (map wrap-var lhs*)])
```

```
           (let ([env (extend-variables env lhs* uvar*)])
             (Letrec
              #f uvar* (map (Expr env safe) rhs*)
              (make-seq `(,((Expr env safe) x) ,(map (Expr env safe) x*) ...))))))]

        [(set! ,id ,[rhs])
         (guard (symbol? id))
         (let ([uvar (lookup id env)])
           (cond
            [(ident? uvar) (ident-assigned-set! uvar #t) (Set #f uvar rhs)]
            [else (App #f (Primref #f #f '$set-top-level-value!)
                       (list (Quote #f uvar) rhs))]))]

        [(apply ,[rator] ,[rand*] ... ,[(Expr env safe) -> rest])
         (App #f (Primref #f #t 'apply) `(,rator ,rand* ... ,rest))]

        ;; application
        [(,[x] ,arg* ...) (Application env x arg*)]

        ;; free identifiers
        [,id
         (guard (symbol? id))
         (begin
           (display (list "WARNING: top-level-value" id))(newline)
           (App #f (Primref #f #f '$top-level-value) (list (Quote #f id))))]

        [,x (assertion-violation who "invalid Expr" x)])))

(define parse-scheme (Expr empty-env #t))
(define parse-unsafe-scheme (Expr empty-env #f))

(define (unparse-scheme x)
  (define (Ref->ident x)
    (Trex-case x
      [(Ref src id) id]
      [else #f]))
  (Trex-case x
    [(Ref src id) (ident-name id)]
    [(Quote src datum) `(quote ,datum)]
    [(Seq src e1 e2) `(begin ,(unparse-scheme e1) ,(unparse-scheme e2))]
    [(CaseLambda src cl*)
     (let ([cl*
            (map (lambda (cl)
                   (Trex-CaseLambdaClause-case
                    cl [(CaseLambdaClause fmls body)
                        `(,(improper-map ident-name fmls) ,(unparse-scheme body))]))
                 cl*)])
       (if (= (length cl*) 1)
           `(lambda ,(caar cl*) ,(cadar cl*))
           `(case-lambda ,@cl*)))]
    [(App src rator rand*)
     `(,(unparse-scheme rator) ,@(map unparse-scheme rand*))]
    [(Letrec src var* expr* body)
     `(letrec ,(map (lambda (var expr)
                      `[,(ident-name var) ,(unparse-scheme expr)]) var* expr*)
        ,(unparse-scheme body))]
    [(If src t c a)
     `(if ,(unparse-scheme t) ,(unparse-scheme c) ,(unparse-scheme a))]
    [(Set src var expr) `(set! ,(ident-name var) ,(unparse-scheme expr))]
```

```
    [(Primref src safe name)
     (if safe
         name
         (string->symbol (string-append "unsafe-" (symbol->string name))))]))

)
```

Listing B.4: tests.ss

```
(library (tests)
  (export run-tests)
  (import (rnrs) (match) (driver-function)
          (only (chezscheme) display-condition pretty-print))

(define (alpha-equiv? x y env)
  (define (alpha-equiv-var? x y)
    (let ([ypair (assq y env)])
      (if ypair (eq? x (cdr ypair)) (eq? x y))))
  (define (alpha-equiv-clause clx cly)
    (match (list clx cly)
      [((,xx* ...) ,xb) ((,yx* ...) ,yb))
       (let ([env (append (map cons yx* xx*) env)])
         (alpha-equiv? xb yb env))]
      [((,xx* ... . ,xr) ,xb) ((,yx* ... . ,yr) ,yb))
       (let ([env (cons (cons yr xr) (append (map cons yx* xx*) env))])
         (alpha-equiv? xb yb env))]))
  (match `(,x ,y)
    [((lambda . ,clx) (lambda . ,cly)) (alpha-equiv-clause clx cly)]
    [((case-lambda ,clx* ...) (case-lambda ,cly* ...))
     (for-all alpha-equiv-clause clx* cly*)]
    [((letrec ([,xx* ,xe*] ...) ,xb) (letrec ([,yx* ,ye*] ...) ,yb))
     (let ([env (append (map cons yx* xx*) env)])
       (and (for-all (lambda (x y) (alpha-equiv? x y env)) xe* ye*)
            (alpha-equiv? xb yb env)))]
    [((set! ,xx ,xv) (set! ,yx ,yv))
     (and (alpha-equiv-var? xx yx) (alpha-equiv? xv yv env))]
    [(,x ,y) (guard (and (symbol? x) (symbol? y))) (alpha-equiv-var? x y)]
    [((begin ,xa ,xb) (begin ,ya ,yb))
     (and (alpha-equiv? xa ya env) (alpha-equiv? xb yb env))]
    [((quote ,x) (quote ,y)) (equal? x y)]
    [((if ,xt ,xc ,xa) (if ,yt ,yc ,ya))
     (and (alpha-equiv? xt yt env)
          (alpha-equiv? xc yc env)
          (alpha-equiv? xa ya env))]
    [((prim-wrap ,xchanged? ,xprim ,xb) (prim-wrap ,ychanged? ,yprim ,yb))
     (and (eq? xprim yprim) (alpha-equiv? xb yb env) (eq? xchanged? ychanged?))]
    [((,xrator ,xrand* ...) (,yrator ,yrand* ...))
     (and (alpha-equiv? xrator yrator env)
          (= (length xrand*) (length yrand*))
          (for-all (lambda (x y) (alpha-equiv? x y env)) xrand* yrand*))]
    [(,x ,y) #f]))

(define tests '(
  [1 . '1]
  [(cons 1 2) . (cons '1 '2)]
  [(car (cons 1 2)) . (unsafe-car (cons '1 '2))]
  [(lambda () 1) . (lambda () '1)]
```

```
[(lambda (x) x) . (lambda (x) x)]
[(lambda x x) . (lambda x x)]
[(lambda (x . y) x) . (lambda (x . y) x)]
[(lambda (x . y) y) . (lambda (x . y) y)]
[(let ([z (lambda (x) x)]) (let ([y (z (cons 1 2))]) (car y))) .
 (letrec ([z (lambda (x) x)]) (letrec ([y (z (cons '1 '2))]) (unsafe-car y)))]
[(let ([z (lambda (f y) (f y))])
   (let ([y (z (lambda (x) x) (cons 1 2))]) (car y))) .
 (letrec ([z (lambda (f y) (f y))])
   (letrec ([y (z (lambda (x) x) (cons '1 '2))]) (unsafe-car y)))]
[(cons #f (case-lambda
           [(x y z) (cons x (cons y z))]
           [(x . r) (cons r x)]
           [q (car q) (cdr q)])) .
 (cons '#f (case-lambda
            [(x y z) (cons x (cons y z))]
            [(x . r) (cons r x)]
            [q (car q)]))]
[(let ([x (cons 1 2)]) (car x)) . (letrec ([x (cons '1 '2)]) (unsafe-car x))]
[(let ([x (cons 1 2)]) (car x) (car x)) .
 (letrec ([x (cons '1 '2)]) (begin (unsafe-car x) (unsafe-car x)))]
[(let ([x (cons 1 2)]) (car x) (vector-length x)) .
 (letrec ([x (cons '1 '2)]) (begin (unsafe-car x) (vector-length x)))]
[(let ([x (read)]) (car x) (car x)) .
 (letrec ([x (read)]) (begin (car x) (unsafe-car x)))]
[(if (error 'who "message") 1 2) . (error 'who '"message")]
[(if #t 1 2) . (begin '#t '1)] ;; test unreachable empty environments
[(let ((f (lambda (x) (if (fxzero? x) 1 2)))) (f 0) (f 3)) .
 (letrec ([f (lambda (x) (if (fxzero? x) '1 '2))]) (begin (f '0) (f '3)))]
;;; Olin Shiver's puzzle test (page 140 (in pdf - 118) of his thesis):
[(let ([f (lambda (x h) (if (fxzero? x) (h) (lambda () x)))]) (f 0 (f 3 #f))) .
 (letrec ([f (lambda (x h) (if (fxzero? x) (h) (lambda () x)))])
   (f '0 (f '3 '#f)))]
[(letrec ([x (cons 1 2)] [f (lambda () (car x))]) (f)) .
 (letrec ([x (cons '1 '2)] [f (lambda () (unsafe-car x))]) (f))]
;; Test that bare var refs properly effect the environment
[(let ([x (if (read) (cons 1 2) '#f)]) (if x (car x) 3)) .
 (letrec ([x (if (read) (cons '1 '2) '#f)]) (if x (unsafe-car x) '3))]
[(let ([x (read)]) (let ([a (car x)]) (car x))) .
 (letrec ([x (read)]) (letrec ([a (car x)]) (unsafe-car x)))]
[(let ([x (read)]) (if (if (pair? x) #f #t) (car x) (car x))) .
 (letrec ([x (read)]) (if (if (pair? x) '#f '#t) (car x) (unsafe-car x)))]
[(letrec ([a (car (cons 1 2))]) 1) . (letrec ([a (unsafe-car (cons '1 '2))]) '1)]
[(letrec ((x (cons 1 2))) (car x)) . (letrec ([x (cons '1 '2)]) (unsafe-car x))]
[(let ([x (cons 1 2)]) (letrec ([a (car x)][b (car x)]) 1)) .
 (letrec ([x (cons '1 '2)]) (letrec ([a (unsafe-car x)][b (unsafe-car x)]) '1))]
;; the cons call to f never happens
[(let ([f (lambda (x) (car x) 1)]) (f (make-vector 3)) (f (cons 1 2))) .
 (letrec ([f (lambda (x) (car x))]) (f (unsafe-make-vector '3)))]
;; same as last test but both calls are made because of different order
[(let ([f (lambda (x) (car x) 1)]) (f (cons 1 2)) (f (make-vector 3)) 4) .
 (letrec ([f (lambda (x) (begin (car x) '1))])
   (begin (f (cons '1 '2)) (f (unsafe-make-vector '3))))]
[(car (read)) . (car (read))]
[(cons 1 2) . (cons '1 '2)]
[(read) . (read)]
[(letrec ([a (car b)][b (car a)]) 3) .
 (letrec ([a (unsafe-car b)][b (unsafe-car a)]) '3)]
[(letrec ([a (lambda () (car b))][b (lambda () (car a))]) 3) .
```

```
 (letrec ([a (case-lambda)] [b (case-lambda)]) '3)]
[(letrec ([a (cons 1 2)] [b (car a)]) 3) .
 (letrec ([a (cons '1 '2)][b (unsafe-car a)]) '3)]
[(let ([a (lambda () (car (read)))]) 1) .
 (letrec ([a (case-lambda)]) '1)]
[(begin (error 'who "message") #t) . (error 'who '"message")]
;; tests that info is gathered from both args of letrec
[(let ([x (read)]) (letrec ([a (car x)][b (vector-length x)]) 4 5)) .
 (letrec ([x (read)]) (letrec ([a (car x)][b (vector-length x)]) '4))]
[(let ([x (lambda () 1 2)]) 3) . (letrec ([x (case-lambda)]) '3)]
[(let ([x (read)]) (if x x (error 'who "message")) (if x 1 2)) .
 (letrec ([x (read)]) (begin (if x x (error 'who '"message")) (begin x '1)))]
[(let ([x (cons 1 2)]) (car x) (set! x 5) (car x)) .
 (letrec ([x (cons '1 '2)]) (begin (begin (car x) (set! x '5)) (car x)))]
[(let ([x (cons 1 2)]) (car x) (set! x (cons 3 4)) (car x)) .
 (letrec ([x (cons '1 '2)])
   (begin (begin (unsafe-car x) (set! x (cons '3 '4))) (unsafe-car x)))]
[(let ([f (lambda (x) (set! x (cons 1 2)) x)]) (car (f (cons 4 5)))) .
 (letrec ([f (lambda (x) (begin (set! x (cons '1 '2)) x))])
   (unsafe-car (f (cons '4 '5))))]
;; The following works, should reach the 5, since (cons - -) is true
[(let ([f (lambda (x y) (cons (car x) (fx+ y 1)))] [x (read)] [y (read)])
   (if (f x y) 5 7)) .
 (letrec ([f (lambda (x y) (cons (car x) (fx+ y '1)))] [x (read)] [y (read)])
   (begin (f x y) '5))]
[(let ([x (read)] [f (lambda (a b) (cdr a))]) (car x) (f x (car x))) .
 (letrec ([x (read)] [f (lambda (a b) (unsafe-cdr a))])
   (begin (car x) (f x (unsafe-car x))))]
[(let ([x (read)][f (lambda (a b) (cdr a))]) (f (begin (car x) x) (car x))) .
 (letrec ([x (read)] [f (lambda (a b) (unsafe-cdr a))])
   (f (begin (car x) x) (car x)))]
[(let ([x (read)]) (if #t (car x) 1) (car x)) .
 (letrec ([x (read)]) (begin (begin '#t (car x)) (unsafe-car x)))]
[(letrec ([x 1]) (set! x (cons 1 2)) (car x)) .
 (letrec ([x '1]) (begin (set! x (cons '1 '2)) (car x)))]
;; check that escaping lambdas work right
[(if (read) (lambda (x) (car x)) (lambda (y) (cdr y))) .
 (if (read) (lambda (x) (car x)) (lambda (y) (cdr y)))]
;; should mark-escaped-lambda
[(cons 1 (lambda (x) (cdr x))) . (cons '1 (lambda (x) (cdr x)))]
;; should not mark-escaped-lambda
[(null? (lambda (x) (cdr x))) . (null? (case-lambda))]
;; test that car never happens due to type error in cons call
[(car (cons 1)) . (unsafe-car (cons '1))]
;; test that low gas works ok
[(let ([x (read)]) (if (not (not (pair? x))) (car x))) .
 (letrec ([x (read)])
   (if (if (if (pair? x) '#f '#t) '#f '#t) (unsafe-car x) (void)))]
[(lambda () (void) 3) . (lambda () (begin (void) '3))]
[(letrec ([fac (lambda (x) (if (fxzero? x) 1 (fx* x (fac 0))))]) (fac (read))) .
 (letrec ([fac (lambda (x) (if (fxzero? x) '1 (unsafe-fx* x (fac '0))))])
   (fac (read)))]
;; test incorrect argument count calls
[((lambda (x) 1 2) 3 4) . ((case-lambda) '3 '4)]
;; too many arguments and we have a last rest (should call body)
[((case-lambda [(x . y) 3 4]) 5 6 7) . ((lambda (x . y) (begin '3 '4)) '5 '6 '7)]
;; too many argument and no last-rest
[((case-lambda [(x y) 3 4]) 5 6 7) . ((case-lambda) '5 '6 '7)]
;; small number of arguments and exact call but no clause handles it
```

```
  [((case-lambda [(x y z w) 3 4]) 5 6 7) . ((case-lambda) '5 '6 '7)]
  ;; small number of argument and exact call and a clause handles it
  [((case-lambda [(x y z w) 3 4]) 5 6 7 8) .
   (letrec ([x '5][y '6][z '7][w '8]) (begin '3 '4))]
  [((lambda (x . y) 3) 1 2) . ((lambda (x . y) '3) '1 '2)]
  ))

(define (run-tests)
  (let ([len (length tests)])
    (let f ([tests tests] [failed '()])
      (cond
       [(null? tests)
        (if (null? failed)
            (begin
              (display "\nran ")
              (display len)
              (display " tests, all succeeded\n"))
            (begin
              (display "\nran ")
              (display len)
              (display " tests, ")
              (display (length failed))
              (display " failed\n")
              (display "Failing tests:\n")
              (for-each pretty-print (reverse failed))))]
       [else
        (let ([fail (lambda ()
                      (display "\nfailed: ")
                      (display (caar tests))
                      (newline)
                      (f (cdr tests) (cons (caar tests) failed)))])
          (guard
           (condition [#t (newline) (display-condition condition) (fail)])
           (let ([result (run (caar tests))])
             (cond
              [(alpha-equiv? (cdar tests) result '())
               (display ".")
               (f (cdr tests) failed)]
              [else (fail)])))))])))

)
```

B.2.1. Preprocessing

Listing B.5: transform-let.ss

```
(library (transform-let)
  (export transform-let)
  (import (rnrs) (tprogram))

;; Trex -> Trex
(define (transform-let trex)
  (define rec transform-let)
```

```
  (or (Trex-case trex
        [(App src-app fun actuals)
         (Trex-case fun
           [(CaseLambda src-lambda clauses)
            (cond
             [(= 1 (length clauses))
              (Trex-CaseLambdaClause-case (car clauses)
                [(CaseLambdaClause formals body)
                 (cond
                  [(and (list? formals) (= (length formals) (length actuals)))
                   (Letrec src-app formals (map rec actuals) (rec body))]
                  [else #f])]
                [else #f])]
             [else #f])]
           [else #f])]
        [else #f])

      (Trex-case trex
        [(Ref src id) trex]
        [(Quote src obj) trex]
        [(Primref src safe name) trex]
        [(Set src id body) (Set src id (rec body))]
        [(App src fun actuals) (App src (rec fun) (map rec actuals))]
        [(If src test thn els) (If src (rec test) (rec thn) (rec els))]
        [(CaseLambda src clauses)
         (let ([f (lambda (clause)
                    (Trex-CaseLambdaClause-case clause
                      [(CaseLambdaClause formals body)
                       (CaseLambdaClause formals (rec body))]))])
           (CaseLambda src (map f clauses)))]
        [(Seq src e1 e2) (Seq src (rec e1) (rec e2))]
        [(Letrec src lhs rhs body) (Letrec src lhs (map rec rhs) (rec body))])))

)
```

Listing B.6: ast-bubbles.ss

```
(library (ast-bubbles)
  (export make-AST-bubbles AST-bubbles-intersect)
  (import (except (rnrs) case cond if) (safe-forms) (myers-stacks)
          (tree-locations) (ast) (idents))

;; KEY TRICK: the bubble algorithm creates the bubble lists in order
;; sorted by their ident-number
(define (AST-bubbles-intersect set1 set2)
  (define < ident>?)
  (define rec AST-bubbles-intersect)
  (cond
   [(or (null? set1) (null? set2)) '()]
   [(< (car set1) (car set2)) (rec (cdr set1) set2)]
   [(< (car set2) (car set1)) (rec set1 (cdr set2))]
   [else (cons (car set1) (rec (cdr set1) (cdr set2)))]))

(define (make-AST-bubbles ast)
  (define tree-location-table
    (make-tree-location-table
     ast
```

```
    AST-tree-start-location-set! AST-tree-end-location-set!
    ident-asts ident-asts-set!
    ident-locations ident-locations-set!
    contains mark add
    (lambda (ast)
      (AST-case ast
        [(Ref ident) (mark ast ident)]
        [(Quote obj) (values)]
        [(Primref safe name) (values)]
        ;; TODO: how to mark non-leaf (for now not needed b/c mutable vars)
        [(Set ident expr) (contains expr)]
        [(App fun args) (contains fun) (vector-for-each contains args)]
        [(If test conseq altern)
         (contains test) (contains conseq) (contains altern)]
        [(CaseLambda table clauses)
         (letrec ([do-clause
                   (lambda (clause)
                     (for-each add (case-lambda-clause-formals&rest clause))
                     (contains (case-lambda-clause-body clause)))])
           (vector-for-each do-clause clauses))]
        [(Seq e1 e2) (contains e1) (contains e2)]
        [(Letrec bindings body)
         (vector-for-each add      (vector-map letrec-binding-lhs bindings))
         (vector-for-each contains (vector-map letrec-binding-rhs bindings))
         (contains body)])
      (values))))

(define (AST-bubble-vars-add! ast ident)
  (AST-bubble-vars-set! ast (cons ident (AST-bubble-vars ast))))

(define (AST-bubble-vars-contains? ast ident)
  (let ([bubbles (AST-bubble-vars ast)])
    (and (not (null? bubbles)) (ident=? ident (car bubbles)))))

(define (bubble-base tpt ast ident)
  (define context (AST-context ast))
  (define locations (ident-locations ident))
  (define asts (ident-asts ident))
  (define prev-ref
    (tree-location-table-prev-ref
     tpt ast ident #f (AST-tree-start-location ast) locations asts))
  (define prev-base
    (myers-stack-eq-suffix
     context (if prev-ref (AST-context prev-ref) myers-stack-null)))
  (define next-ref
    (tree-location-table-next-ref
     tpt ast ident #f (AST-tree-end-location ast) locations asts))
  (define next-base
    (myers-stack-eq-suffix
     context (if next-ref (AST-context next-ref) myers-stack-null)))

  (if (> (myers-stack-length prev-base) (myers-stack-length next-base))
      prev-base
      next-base))

;; inflate - move up context and stop just before would
;;           include extra ref to ident
(define (inflate queue ast ident)
  (let ([base (bubble-base tree-location-table ast ident)]) ;; length >= 0
```

```
      ;; TODO: if not found (then add nothing)
      ;; (instead of or in additon to "if stack-null-base")
      (if (myers-stack-null? base)
          queue
          (let* ([context (AST-context ast)] ;; length >= 1
                 [root (myers-stack-car (myers-stack-previous context base))])
            (if (AST-bubble-vars-contains? root ident)
                queue
                (begin
                  ;; mark root
                  (AST-bubble-vars-add! root ident)
                  ;; add parent of root to queue
                  (cons (myers-stack-car base) queue)))))))

  ;; Give the variables a number so we can keep them sorted
  (let loop ([counter 0]
             [idents (tree-location-table-key-list tree-location-table)])
    (unless (null? idents)
      (ident-number-set! (car idents) counter)
      (loop (+ 1 counter) (cdr idents))))

  ;; NOTE: each ident is separate passes so AST-bubble-vars-add!
  ;; keeps the list uniqified
  (for-each
   (lambda (ident)
     (when (not (ident-assigned ident))
           ;; put all occurances in the work queue
           (let loop ([queue (vector->list (ident-asts ident))])
             (unless (null? queue) ;; loop until queue empty
               (loop (inflate (cdr queue) (car queue) ident))))))
   ;; order is critical to get the sets ordered
   (tree-location-table-key-list tree-location-table)))

)
```

Listing B.7: tree-locations.ss

```
(library (tree-locations)
  (export
   make-tree-location-table
   tree-location-table-key-list
   tree-location-table-prev-ref
   tree-location-table-next-ref)
  (import (rnrs))

(define-record-type
  (tree-location-table $make-tree-location-table tree-location-table?)
  (nongenerative)
  (fields
   key-locations ;; key -> vector[node]
   key-nodes ;; key -> vector[node]
   key-list ;; list[key]
   ))

;; walker must call (for-each contains children) and (mark key)

;; tree-location ::
```

```
;;   obj * (contains * mark * obj -> void) -> tree-location-table
;; mark :: key -> void
;; contains :: obj -> void

(define-syntax make-tree-location-table
  (syntax-rules ()
    [(_ tree ;; node
        node-start-location-set! ;; node * int -> void
        node-end-location-set! ;; node * int -> void
        key-nodes ;; key -> list or vector of node
        key-nodes-set! ;; key * list or vector of node -> void
        key-locations ;; key -> list or vector of int
        key-locations-set! ;; key * list or vector of int -> void
        contains mark add
        walker-fun)

     (let ([counter 0]
           [key-list '()])
       (letrec ([walker walker-fun]
                [contains (lambda (node)
                            (node-start-location-set! node counter)
                            (set! counter (+ 1 counter))
                            (walker node)
                            (set! counter (+ 1 counter))
                            (node-end-location-set! node counter))]
                [mark
                 (lambda (node key)
                   (key-nodes-set!     key (cons node    (key-nodes key)))
                   (key-locations-set! key (cons counter (key-locations key))))]
                [add (lambda (key) (set! key-list (cons key key-list)))])

         ;; walk the tree
         (contains tree)

         ;; Reverse and convert to vectors so we can vector-binary-search
         (for-each
          (lambda (key)
            (key-nodes-set!     key (list->vector (reverse (key-nodes     key))))
            (key-locations-set! key (list->vector (reverse (key-locations key)))))
          key-list)

         ($make-tree-location-table key-locations key-nodes key-list)))]))

(define (key-locations<=? ref key) (<= ref key))

;; returns index of first greater or equal
(define vector-binary-search
  (case-lambda
   [(vector key) (vector-binary-search vector key 0 (vector-length vector))]
   [(vector key start len)
    (define rec vector-binary-search)
    (let* ([len1 (div len 2)]
           [mid (+ start len1)]
           [len2 (- len len1 1)])
      (cond
       [(= len 0) start]
       [(key-locations<=? key (vector-ref vector mid)) (rec vector key start len1)]
       [else (rec vector key (+ mid 1) len2)]))]))
```

```
(define (tree-location-table-prev-ref tpt node key default start-location
                                      key-locations key-nodes)
  (define index (vector-binary-search key-locations start-location))
  (if (= index 0)
      default
      (vector-ref key-nodes (- index 1))))

(define (tree-location-table-next-ref tpt node key default end-location
                                      key-locations key-nodes)
  (define index (vector-binary-search key-locations end-location))
  (if (= index (vector-length key-locations))
      default
      (vector-ref key-nodes index)))

)
```

Listing B.8: myers-stacks.ss

```
(library (myers-stacks)

  ;; PROPERTIES:
  ;;  All operations are O(lg n) or better
  ;;   (except list->myers-stack and myers-stack->list).
  ;;
  ;;  All operations are observationally equivalent to the corresponding
  ;;   list operations.  This includes pointer equality and mutation.

  ;; NOTATION:
  ;;  obj - any scheme object
  ;;  stack - myers-stack object
  ;;  stack-pair - myers-stack object that is not null
  ;;  elem - object contained in the myers-stack
  ;;  list - list of elem objects
  ;;  length - non-negative integer less then or equal to length
  ;;  index - non-negative integer strictly less than length

  (export
   myers-stack
   myers-stack? ;; obj -> boolean

   myers-stack-null ;; stack
   myers-stack-null? ;; obj -> boolean

   myers-stack-cons ;; elem * stack -> stack
   myers-stack-pair? ;; obj -> boolean
   myers-stack-car ;; stack-pair -> elem
   myers-stack-car-set! ;; stack-pair * elem -> void
   myers-stack-cdr ;; stack-pair -> stack

   myers-stack-length ;; stack -> length
   myers-stack-ref ;; stack * index -> elem
   list->myers-stack ;; list -> stack
   myers-stack->list ;; stack -> list

   myers-stack-find ;; (stack ... -> bool) * stack ... -> stack
                    ;; returns first that is true
                    ;; stacks must be same length
```

```
    myers-stack-suffix ;; stack * length -> stack
    myers-stack-drop ;; stack * length -> stack

    myers-stack-previous ;; stack * stack -> stack (second is suffix)
    myers-stack-previous-ref ;; stack * stack -> elem (second is suffix)
    myers-stack-eq-suffix ;; stack * stack -> stack
                          ;; returns first that is 'eq?'
                          ;; stacks need not be same length
    )

  (import (except (rnrs) case cond if) (safe-forms) (records-io))

;; TODO:
;;  - use fx or not?
;;  - fold-right in O(lg n)
;;  - myers-stack-find is only valid if args are same length
;;      (maybe call it find-from-rear and trim arguments to same length)

;;;;;;;;;;;;;;;;
;; Datatype
;;;;;;;;;;;;;;;;

;; Defines the following exported functions:
;;  myers-stack?
;;  myers-stack-car
;;  myers-stack-car-set!
;;  myers-stack-cdr
;;  myers-stack-length
(define-record-type myers-stack
  (nongenerative)
  (fields (mutable $car) length $cdr jump))

(define myers-stack-car-set! myers-stack-$car-set!)

(define (myers-stack-car stack)
  (assert (< 0 (myers-stack-length stack)))
  (myers-stack-$car stack))

(define (myers-stack-cdr stack)
  (assert (< 0(myers-stack-length stack)))
  (myers-stack-$cdr stack))

;;;;;;;;;;;;;;;;
;; Basic Ops
;;;;;;;;;;;;;;;;
(define myers-stack-null
  (let* ([jump-jump (make-myers-stack #f -1 #f #f)]
         [jump (make-myers-stack #f 0 #f jump-jump)])
    (make-myers-stack #f 0 #f jump)))

(define (myers-stack-null? stack)
  (and (myers-stack? stack) (fxzero? (myers-stack-length stack))))

(define (myers-stack-cons car cdr)
  (let* ([jump (myers-stack-jump cdr)]
         [jump-jump (myers-stack-jump jump)])
    (let ([cdr-length (myers-stack-length cdr)]
          [jump-length (myers-stack-length jump)]
          [jump-jump-length (myers-stack-length jump-jump)])
```

```
      (let ([length (fx+ 1 cdr-length)])
        (if (eq? (fx- cdr-length jump-length)
                 (fx- jump-length jump-jump-length))
            (make-myers-stack car length cdr jump-jump)
            (make-myers-stack car length cdr cdr))))))

(define (myers-stack-pair? stack)
  (and (myers-stack? stack) (not (fxzero? (myers-stack-length stack)))))

(define (list->myers-stack list)
  (fold-right myers-stack-cons myers-stack-null list))

(define (myers-stack->list stack)
  (cond
    [(myers-stack-null? stack) '()]
    [(myers-stack-pair? stack)
     (cons (myers-stack-car stack)
           (myers-stack->list (myers-stack-cdr stack)))]))

(define (myers-stack-ref stack index)
  ($check-index 'myers-stack-ref stack index)
  (myers-stack-car (myers-stack-drop stack (- index 1))))

;;;;;;;;;;;;;;;;
;; Basic Finds
;;;;;;;;;;;;;;;;

(define ($check-range who stack index assertion)
  (unless assertion
    (assertion-violation who "out␣of␣range" index stack)))
(define ($check-index who stack index)
  ($check-range who stack index (< index (myers-stack-length stack))))
(define ($check-length who stack length)
  ($check-range who stack length (<= length (myers-stack-length stack))))

;; TODO: what to do if nothing matches the find?
;; TODO: rename to find-first
;; TODO: returns #f if nothing matches
(define (myers-stack-find f . args)
  (cond
    [(apply f args) (apply values args)]
    [(myers-stack-null? (car args)) #f]
    [else
     (let ([jumps (map myers-stack-jump args)])
       (cond
         ;; jump is too far
         [(apply f jumps)
          (apply myers-stack-find f (map myers-stack-cdr args))]
         ;; jump isn't too far
         [else (apply myers-stack-find f jumps)]))]))

(define-syntax myers-stack-find-macro
  (lambda (stx)
    (syntax-case stx ()
      [(_ fun x0 xs ...)
       (with-syntax ([(arg0 args ...) (generate-temporaries #'(x0 xs ...))]
                     [(jump0 jumps ...) (generate-temporaries #'(x0 xs ...))])
         #'(let loop ([f fun][arg0 x0][args xs] ...)
             (cond
```

```
                [(f arg0 args ...) (values arg0 args ...)]
                [(myers-stack-null? arg0) #f]
                [else
                 (let ([jump0 (myers-stack-jump arg0)]
                       [jumps (myers-stack-jump args)] ...)
                   (cond
                     [(f jump0 jumps ...) (loop f (myers-stack-cdr arg0) (
                         myers-stack-cdr args) ...)]
                     [else (loop f jump0 jumps ...)]))])))]))))

(define (myers-stack-suffix stack length)
  ($check-length 'myers-stack-suffix stack length)
  (myers-stack-find-macro
   (lambda (stack) (<= (myers-stack-length stack) length)) stack))

(define (myers-stack-drop stack length)
  ($check-length 'myers-stack-drop stack length)
  (myers-stack-suffix stack (- (myers-stack-length stack) length)))

;;;;;;;;;;;;;;;;
;; Fancy Finds
;;;;;;;;;;;;;;;;

(define (myers-stack-previous stack suffix)
  (myers-stack-suffix stack (+ (myers-stack-length suffix) 1)))

(define (myers-stack-previous-ref stack suffix)
  (and (< (myers-stack-length stack) (myers-stack-length suffix))
       (myers-stack-car (myers-stack-previous stack suffix))))

(define (myers-stack-eq-suffix arg0 arg1)
  (let ([length (min (myers-stack-length arg0)
                     (myers-stack-length arg1))])
    (let-values ([(common0 common1)
                  (myers-stack-find-macro
                   eq?
                   (myers-stack-suffix arg0 length)
                   (myers-stack-suffix arg1 length))])
        common0)))

;;;;;;;;;;;;;;;;
;; Display
;;;;;;;;;;;;;;;;
(record-writer
  (record-type-descriptor myers-stack)
  (lambda (r p wr)
    (display "#[" p)
    (display (record-type-name (record-rtd r)) p)
    (display "␣" p)
    (wr (myers-stack->list r) p)
    (display "]" p)))

)
```

Listing B.9: composition-stacks.ss

```
(library (composition-stacks)
```

```
;; PROPERTIES:
;;  All operations are O(lg n) or better
;;   (except list->composition-stack and composition-stack->list).
;;
;;  All operations are observationally equivalent to the corresponding
;;   list operations.  This includes pointer equality and mutation.

;; NOTATION:
;;  obj - any scheme object
;;  stack - myers-stack object
;;  stack-pair - myers-stack object that is not null
;;  elem - object contained in the myers-stack
;;  list - list of elem objects
;;  length - non-negative integer less then or equal to length
;;  index - non-negative integer strictly less than length

(export
 composition-stack? ;; obj -> boolean

 make-composition-stack-null ;; (elem * elem -> elem) -> stack -- associative
 composition-stack-null? ;; obj -> boolean

 composition-stack-cons ;; elem * stack -> stack
 composition-stack-pair? ;; obj -> boolean
 composition-stack-car ;; stack-pair -> obj
 ;; composition-stack-car-set! ;; stack-pair * obj -> void
 composition-stack-cdr ;; stack-pair -> stack

 composition-stack-length ;; stack -> length
 composition-stack-ref ;; stack * index -> elem
 ;; composition-stack-set! ;; stack * index * elem -> void
 list->composition-stack ;; list -> stack
 composition-stack->list ;; stack -> list

 composition-stack-find ;; (stack ... -> bool) * stack ... -> stack
                        ;; returns first that is true
                        ;; stacks must be same length
 composition-stack-suffix ;; stack * length -> stack
 composition-stack-drop ;; stack * length -> stack

 composition-stack-previous ;; stack * stack -> stack (second is suffix)
 composition-stack-previous-ref ;; stack * stack -> elem (second is suffix)
 composition-stack-eq-suffix ;; stack * stack -> stack
                             ;; returns first that is 'eq?'
                             ;; need not be same length
 composition-stack-range ;; elem * stack * stack -> elem
 )

(import (rnrs)
        (rename
         (myers-stacks)
         (myers-stack-null? composition-stack-null?)
         (myers-stack-pair? composition-stack-pair?)
         (myers-stack-car composition-stack-car)
         ;; myers-stack-car-set! is not exported (b/c it breaks the compositions)
         (myers-stack-cdr composition-stack-cdr)
         (myers-stack-length composition-stack-length)
         (myers-stack-ref composition-stack-ref)
```

```
          (myers-stack->list composition-stack->list)
          (myers-stack-find composition-stack-find)
          (myers-stack-suffix composition-stack-suffix)
          (myers-stack-drop composition-stack-drop)
          (myers-stack-previous composition-stack-previous)
          (myers-stack-previous-ref composition-stack-previous-ref)
          (myers-stack-eq-suffix composition-stack-eq-suffix)))

;; TODO:
;;  - make car non-mutable
;;  - calculate constant for composition-stack-jump

;;;;;;;;;;;;;;;;
;; Datatype
;;;;;;;;;;;;;;;;

(define-record-type composition-stack
  (nongenerative)
  (fields composer composite)
  (parent myers-stack))

(define composition-stack-jump
  (let ([rtd (record-type-descriptor myers-stack)]
        [index 3])
    (assert (eq? 'jump (vector-ref (record-type-field-names rtd) index)))
    (record-accessor rtd index)))

;;;;;;;;;;;;;;;;
;; Operations
;;;;;;;;;;;;;;;;
(define (make-composition-stack-null composer)
  (let* ([jump-jump (make-composition-stack #f -1 #f #f #f #f)]
         [jump (make-composition-stack #f 0 #f jump-jump #f #f)])
    (make-composition-stack #f 0 #f jump composer #f)))

(define (composition-stack-cons car cdr)
  (let* ([jump (composition-stack-jump cdr)]
         [jump-jump (composition-stack-jump jump)])
    (let ([cdr-length (composition-stack-length cdr)]
          [jump-length (composition-stack-length jump)]
          [jump-jump-length (composition-stack-length jump-jump)])
      (let ([length (fx+ 1 cdr-length)]
            [composer (composition-stack-composer cdr)])
        (if (eq? (fx- cdr-length jump-length)
                 (fx- jump-length jump-jump-length))
            (make-composition-stack
             car length cdr jump-jump composer
             (composer car (composer (composition-stack-composite cdr)
                                     (composition-stack-composite jump))))
            (make-composition-stack
             car length cdr cdr composer car))))))

(define (list->composition-stack compose list)
  (fold-right
   composition-stack-cons (make-composition-stack-null compose) list))

(define (composition-stack-range identity stack suffix)
  (define (rec identity composer stack length)
    (if (fx=? length (composition-stack-length stack))
```

```
        identity
        (let ([jump (composition-stack-jump stack)])
          (if (fx>? length (composition-stack-length jump))
              (composer
               (composition-stack-car stack)
               (rec identity composer (composition-stack-cdr stack) length))
              (composer
               (composition-stack-composite stack)
               (rec identity composer jump length))))))
  (rec identity (composition-stack-composer stack)
       stack (composition-stack-length suffix)))

)
```

B.2.2. Graph construction

Listing B.10: env-flow-graphs.ss

```
(library (env-flow-graphs)
  (export
   make-env-flow-graph ;; ast -> void
   env-flow-graph-force! ;; ast -> void
   )
  (import (except (rnrs) case cond if) (safe-forms) (composition-stacks)
          (env-flow-functions) (ast) (ast-flow-graphs) (types) (flow-graphs))

;;;;;;;;;;;;;;;;
;; defines transfer function from output of child to output of this
;; thus does one recur per child

(define (make-env-flow-graph ast)
  ;; context :: composition-stack of flow-graph-node of env-flow-function
  (define (rec ast context)

    (define-syntax define-rec
      (syntax-rules ()
        [(_ (name args ...)
            ([true-test true-conseq] ...)
            ([false-test false-conseq] ...))
         (define (name ast context expr args ...)
           (define node
             (make-AST-node ast make-flow-graph-lazy-node (symbol->string 'name)
                            env-flow-function=? env-flow-function-zero))
           (AST-graph-context-thunk-set!
            expr (lambda ()
                   (flow-graph-lazy-node-force!
                    node
                    (lambda (args ...)
                      (cond-env-flow-function
                       ([true-test true-conseq] ...)
                       ([false-test false-conseq] ...)))
                    (AST-output-node args) ...)))
           (rec expr (composition-stack-cons node context)))])))
```

```
    (define-rec (rec-id)
      ([#t true])
      ([#t false]))

    (define-rec (rec-bypass)
      ([#t bypass])
      ([#t bypass]))

    (define-rec (rec-void ast)
      ([(true-type? ast) true]  [(true-type? ast) false])
      ([(false-type? ast) true] [(false-type? ast) false]))

    (define-rec (rec-if-test conseq altern)
      ([(true-type? conseq) true]  [(true-type? altern) false])
      ([(false-type? conseq) true] [(false-type? altern) false]))

    (define-rec (rec-if-branch this that)
      ([#t true]  [(true-type? that) bypass])
      ([#t false] [(false-type? that) bypass]))

    (AST-graph-context-set! ast context)

    (AST-case ast
      [(Ref var) (values)]
      [(Quote obj) (values)]
      [(Primref safe name) (values)]
      [(Set var expr) (rec-void ast context expr ast)]

      [(App fun actuals)
       (rec-void ast context fun ast)
       (vector-for-each (lambda (actual) (rec-void ast context actual ast))
                        actuals)]

      [(If test conseq altern)
       (rec-if-test ast context test conseq altern)
       (rec-if-branch ast context conseq conseq altern)
       (rec-if-branch ast context altern altern conseq)]

      [(CaseLambda table clauses)
       (vector-for-each
        (lambda (clause)
          ;; NOTE: context here is only used for in not out
          (rec-bypass ast context (case-lambda-clause-body clause)))
        clauses)]

      [(Seq expr1 expr2)
       (rec-void ast context expr1 ast)
       (rec-id ast context expr2)]

      [(Letrec bindings body)
       (vector-for-each
        (lambda (binding) (rec-void ast context (letrec-binding-rhs binding) ast))
        bindings)
       (rec-id ast context body)]))

  (rec ast (make-composition-stack-null
            (lambda (inner outer)
              (make-AST-node ast make-flow-graph-function-node
                             "env-flow-function-compose"
```

```
                              env-flow-function=?
                              env-flow-function-compose outer inner)))))

(define (env-flow-graph-force! ast)
  (define (rec ast) (env-flow-graph-force! ast))
  (let ([thunk (AST-graph-context-thunk ast)])
    (when thunk
      (thunk)
      (AST-graph-context-thunk-set! ast #f)))

  (AST-case ast
    [(Ref var) (values)]
    [(Quote obj) (values)]
    [(Primref safe name) (values)]
    [(Set var expr) (rec expr)]
    [(App fun actuals)
     (rec fun)
     (vector-for-each rec actuals)]
    [(If test conseq altern)
     (rec test)
     (rec conseq)
     (rec altern)]
    [(CaseLambda table clauses)
     (vector-for-each (lambda (clause) (rec (case-lambda-clause-body clause)))
                      clauses)]
    [(Seq expr1 expr2)
     (rec expr1)
     (rec expr2)]
    [(Letrec bindings body)
     (vector-for-each (lambda (binding) (rec (letrec-binding-rhs binding)))
                      bindings)
     (rec body)]))

)
```

Listing B.11: value-flow-graphs.ss

```
(library (value-flow-graphs)
  (export make-value-flow-graph clause-info-in-called)
  (import (except (rnrs) case cond if) (safe-forms) (define-ignored) (ast)
          (ast-flow-graphs) (flow-graphs) (types) (bitypes) (idents)
          (composition-stacks) (env-flow-functions) (prims) (iota) (ast-bubbles))

(define make-AST-for-all-node
  (case-lambda
   [(ast name fun base nodes1 nodes2)
    (cond
     [(null? nodes1) base]
     [else
      (make-AST-for-all-node
       ast name fun (make-AST-node ast make-flow-graph-function-node name boolean=?
                                   fun base (car nodes1) (car nodes2))
       (cdr nodes1) (cdr nodes2))])]

   [(ast name fun base nodes)
    (cond
     [(null? nodes) base]
```

```
     [else
      (make-AST-for-all-node
       ast name fun (make-AST-node ast make-flow-graph-function-node name boolean=?
                                   fun base (car nodes))
       (cdr nodes))])]))

(define make-AST-for-all-non-bot-node
  (case-lambda
   [(ast name nodes)
    (cond
     [(null? nodes) (make-AST-const-node ast name #t)]
     [else (make-AST-for-all-non-bot-node
            ast name (make-AST-node ast make-flow-graph-function-node name boolean=?
                                    non-bot-type? (car nodes))
            (cdr nodes))])]
   [(ast name base nodes)
    (cond
     [(null? nodes) base]
     [else
      (make-AST-for-all-non-bot-node
         ast name (make-AST-node
                   ast make-flow-graph-function-node name boolean=?
                   (lambda (base node) (and base (non-bot-type? node)))
                   base (car nodes))
         (cdr nodes))])]))

(define (make-AST-const-node ast name const)
  (make-AST-node ast make-flow-graph-mutable-node name eq? const))
(define (make-AST-type-guard-node ast name guard node)
  (make-AST-node ast make-flow-graph-function-node name type=?
                 (lambda (guard val) (if guard val bot-type)) guard node))
(define (make-AST-const-type-guard-node ast name guard const)
  (make-AST-node ast make-flow-graph-function-node name type=?
                 (lambda (test) (if test const bot-type)) guard))

(define (make-AST-bitype-id-node ast name node)
  (make-AST-node ast make-flow-graph-function-node name bitype=?
                 (lambda (x) x) node))
(define (AST->type-id-node ast name expr)
  (AST-output-node expr))

(define (make-AST-enable-node ast name)
  (make-AST-node ast make-flow-graph-mutable-node name boolean=? #f))
(define (bool-node-enable! node) (flow-graph-mutable-node-set! node #t))

(define (make-AST-binding-node ast name)
  (make-AST-node ast make-flow-graph-mutable-node name type=? bot-type))

(define (make-AST-var-binding-node ast name var)
  (let ([binding-node
         (make-AST-binding-node
          ast (string-append name ":" (symbol->string (ident-symbol-name var))))])
    (ident-binding-node-set! var binding-node)
    (ident-preceding-ref-set! var (make-down-ref binding-node))
    binding-node))

(define (type-node-add! node value)
  (flow-graph-mutable-node-set!
   node (type-union value (flow-graph-node-output node))))
```

```
80
(define (make-AST-binding-node-add! ast name var node)
  (make-AST-node
   ast make-flow-graph-function-node name eq?
   (lambda (x) (type-node-add! (ident-binding-node var) x)) node))
85
(define (AST->bool-node ast name function expr)
  (make-AST-node ast make-flow-graph-function-node name boolean=? function
                 (AST-output-node expr)))

90 (define-record-type clause-info
  (nongenerative)
  (fields has-escaped in-called in-formals in-rest
          out-return out-return:rest-is-null out-return:rest-is-pair
          out-formals out-formals-bitop)
95  (protocol
   (lambda (make)
     (lambda (clause
              has-rest has-escaped
              in-called in-formals&rest
100           out-return out-formals&rest out-formals-bitop)

       (define (split-formals&rest list)
         (define (rec list)
           (cond
105         [(null? (cdr list)) (values '() (car list))]
            [else
             (let-values ([(formals rest) (rec (cdr list))])
               (values (cons (car list) formals) rest))]))
         (cond
110       [(not has-rest) (values list #f)]
          [else (rec list)]))

       (let-values
           ([(in-formals in-rest) (split-formals&rest in-formals&rest)]
115         [(out-formals out-rest) (split-formals&rest out-formals&rest)])

         (define (make-out-return name pred?)
           (define (rest-return-type rest-bitype return-type)
             (type-filter
120           return-type
              (pred? (bitype-true rest-bitype))
              (pred? (bitype-false rest-bitype))))
           (if (not out-rest)
               out-return
125            (make-AST-node clause
                              make-flow-graph-function-node name type=?
                              rest-return-type out-rest out-return)))
         (make
             has-escaped in-called (list->vector in-formals) in-rest
130          out-return
             (make-out-return "out-return&rest-is-null" null-type?)
             (make-out-return "out-return&rest-is-pair" pair-type?)
             (list->vector out-formals) out-formals-bitop))))))

135 (define-record-type up-ref (nongenerative) (fields ast node))
(define-record-type down-ref (nongenerative) (fields node))

(define-syntax define-proc-fun
```

```
  (syntax-rules ()
    [(_ (name clause-info formal-count arg-count args ...) body ...)
     (define (name table clauses missing-clause-node)
       (lambda (arg-count args ...)
         (let ([clause (case-lambda-clause-ref table clauses arg-count)])
           (if (not clause)
               missing-clause-node
               (let ([clause-info (case-lambda-clause-info clause)])
                 (let ([formal-count
                        (vector-length
                         (clause-info-out-formals clause-info))])
                   (assert (>= arg-count formal-count))
                   ;; clause-info formal-count arg-count args
                   body ...)))))))]))

;; procedures for proc-type
(define (escape-proc has-escaped-node)
  (lambda () (bool-node-enable! has-escaped-node)))

(define-proc-fun (input-called-proc
                  clause-info formal-count arg-count callable)
  (when callable
    (let ([in-rest (clause-info-in-rest clause-info)])
      (when in-rest
        (type-node-add! in-rest
                        (cond
                         [(= arg-count formal-count) (type-of '())]
                         [(> arg-count formal-count) (type-of '(() . ()))])))))
    (bool-node-enable! (clause-info-in-called clause-info))))

(define-proc-fun (input-argument-proc
                  clause-info formal-count arg-count arg-position argument-type)
  (if (>= arg-position formal-count)
      (begin
        ;; values escape that go into rest arguments
        (assert (clause-info-in-rest clause-info))
        (mark-escaped-type! argument-type)
        '() ;; must return a value
        )
      (type-node-add!
       (vector-ref (clause-info-in-formals clause-info) arg-position)
       argument-type)))

(define-proc-fun (output-return-proc clause-info formal-count arg-count)
  (cond
   [(= arg-count formal-count) (clause-info-out-return:rest-is-null clause-info)]
   [(> arg-count formal-count) (clause-info-out-return:rest-is-pair clause-info)]))

(define-proc-fun (output-argument-proc
                  clause-info formal-count arg-count arg-position)
  (if (< arg-position formal-count)
      (vector-ref (clause-info-out-formals clause-info) arg-position)
      (clause-info-out-formals-bitop clause-info)))

;; AST * a -> ident or a
(define (AST-Ref->ident ast default)
  (AST-case ast
    [(Ref ident) ident]
    [else default]))
```

```
;; AST -> node (output-node or preceding-ref-node)
(define (AST->bare-var-node ast)
  (define var (AST-Ref->ident ast #f))
  ;; feed back if (1) it is a var and (2) it is an intersected
  ;; var, i.e., don't feed back singlely mentioned variables

  ;; NOTE: memq is O(1) b/c on a var the bubble length is at most one
  (if (and var (memq var (AST-bubble-vars ast)))
      (down-ref-node (ident-preceding-ref var))
      (AST-output-node ast)))

;; TODO: define turn-var-up/down in terms of single function
;; function must return a type
(define (turn-var-up! ast name function var)
  (define preceding-ref (ident-preceding-ref var))
  (define preceding-node (down-ref-node preceding-ref))
  (define node-name (string-append "turn-var-up:" name))
  (ident-preceding-ref-set!
   var (make-up-ref ast (make-AST-node ast make-flow-graph-function-node node-name
                                       bitype=? function preceding-node))))

;; function must return a bitype
(define (turn-var-down! ast name function var)
  (define preceding-ref (ident-preceding-ref var))
  (define preceding-node (up-ref-node preceding-ref))
  (define node-name (string-append "turn-var-down:" name ":"
                                   (symbol->string (ident-symbol-name var))))
  (ident-preceding-ref-set!
   var (make-down-ref (make-AST-node
                       ast make-flow-graph-function-node
                       node-name type=? function preceding-node))))

(define-syntax parallel-loop
  (syntax-rules ()
    [(_ ast-lists arguments action)
     (let loop ([asts ast-lists][args arguments][posts '()])
       (cond
        [(null? args) posts]
        [else
         (let* ([vars (AST-bubble-vars (car asts))]
                [pre (map* ident-preceding-ref vars)]
                [result (action (car args))] ;; ignore result
                [post (map* ident-preceding-ref vars)])
           (for-each* ident-preceding-ref-set! vars pre)
           (loop (cdr asts) (cdr args) (cons post posts)))]))]))

(define-syntax list-rev
  (syntax-rules ()
    [(_ acc) acc]
    [(_ acc x xs ...) (list-rev (cons x acc) xs ...)]))

(define-syntax parallel-list
  (syntax-rules ()
    [(_ (asts actions) ...)
     (list-rev
      '()
      (let* ([vars (AST-bubble-vars asts)]
             [pre (map* ident-preceding-ref vars)]
```

```
             [result actions] ;; ignore result
             [post (map* ident-preceding-ref vars)])
          (for-each* ident-preceding-ref-set! vars pre)
          post) ...)]))

(define-syntax fold-left* ;; equivalent to fold-left but ensures inlining
  (syntax-rules ()
    [(_ fun base args)
     (let loop ([b base]
                [a args])
       (cond
        [(null? a) b]
        [else (loop (fun b (car a)) (cdr a))]))]
    [(_ fun base args1 args2)
     (let loop ([b base]
                [a1 args1]
                [a2 args2])
       (cond
        [(null? a1) b]
        [else (loop (fun b (car a1) (car a2)) (cdr a1) (cdr a2))]))]))

(define-syntax for-each* ;; equivalent to for-each but ensures inlining
  (syntax-rules ()
    [(_ fun args)
     (let loop ([a args])
       (cond
        [(null? a) (values)]
        [else (fun (car a)) (loop (cdr a))]))]
    [(_ fun args1 args2)
     (let loop ([a1 args1]
                [a2 args2])
       (cond
        [(null? a1) (values)]
        [else (fun (car a1) (car a2)) (loop (cdr a1) (cdr a2))]))]
    [(_ fun args1 args2 args3)
     (let loop ([a1 args1]
                [a2 args2]
                [a3 args3])
       (cond
        [(null? a1) (values)]
        [else (fun (car a1) (car a2) (car a3))
              (loop (cdr a1) (cdr a2) (cdr a3))]))]))

(define-syntax map* ;; equivalent to map but ensures inlining
  (syntax-rules ()
    [(_ fun args)
     (let loop ([a args])
       (cond
        [(null? a) '()]
        [else (cons (fun (car a)) (loop (cdr a)))]))]
    [(_ fun args1 args2)
     (let loop ([a1 args1]
                [a2 args2])
       (cond
        [(null? a1) '()]
        [else (cons (fun (car a1) (car a2)) (loop (cdr a1) (cdr a2)))]))]))

(define-syntax parallel*
  (syntax-rules ()
```

```
    [(_ parent-ast* ident-up? node-up? eqv? first-ref combine asts action)
     (let ()
       (define parent-ast parent-ast*)

       ;; Get post values
       (define posts (reverse action))

       ;; Mark vars as not filled with a new value yet and collect the idents
       (define idents
         (fold-left*
          (lambda (idents ast)
            (fold-left*
             (lambda (idents ident)
               (if (ident-preceding-ref ident)
                   (begin
                     (ident-preceding-ref-set! ident #f)
                     (cons ident idents))
                   idents))
             idents
             (AST-bubble-vars ast)))
          '()
          asts))

       ;; Fill in vars & return list of uniq vars
       (for-each* (lambda (ast post)
                    (for-each*
                     (lambda (ident node)
                       (ident-preceding-ref-set!
                        ident
                        (cond
                          [(not combine) node]
                          [(not (ident-preceding-ref ident)) (first-ref node)]
                          [else
                           (let ([node (make-AST-node
                                        parent-ast make-flow-graph-function-node
                                        "parallel-uniq" eqv? combine
                                        ((if ident-up? up-ref-node down-ref-node)
                                         (ident-preceding-ref ident))
                                        ((if node-up? up-ref-node down-ref-node)
                                         node))])
                             (if ident-up?
                                 (make-up-ref parent-ast node)
                                 (make-down-ref node)))])))
                     (AST-bubble-vars ast) post))
                  asts posts)
       idents)]))

(define (parallel parent-ast up? eqv? first-ref combine asts thunks)
  (define (parallel-one ast thunk)
    (define vars (AST-bubble-vars ast))
    (define pre (map* ident-preceding-ref vars)) ;; save inner var nodes
    (define thunk-result (thunk)) ;; ignore result
    (define post (map* ident-preceding-ref vars)) ;; get inner var nodes
    (for-each* ident-preceding-ref-set! vars pre) ;; restore inner vars
    post)
  (define (add-node! ident node)
    (define field (if (eq? up? #t) up-ref-node down-ref-node))
    (ident-preceding-ref-set!
     ident
```

```
    (if (not (ident-preceding-ref ident))
        (first-ref node)
        (let ([node (make-AST-node
                     parent-ast make-flow-graph-function-node
                     "parallel-uniq" eqv? combine
                     (field (ident-preceding-ref ident))
                     ((if (eq? up? 'mixed)
                          up-ref-node field) node))])
          (if (eq? up? #t)
              (make-up-ref parent-ast node)
              (make-down-ref node))))))

  ;; Get post values
  (define posts (map* parallel-one asts thunks))

  ;; Mark vars as not filled with a new value yet and collect the idents
  (define idents
    (fold-left*
     (lambda (idents ast)
       (fold-left*
        (lambda (idents ident)
          (if (ident-preceding-ref ident)
              (begin
                (ident-preceding-ref-set! ident #f)
                (cons ident idents))
              idents))
        idents
        (AST-bubble-vars ast)))
     '()
     asts))

  ;; Fill in vars & return list of uniq vars
  (for-each* (lambda (ast post) (for-each* add-node! (AST-bubble-vars ast) post))
             asts posts)
  idents)

;; Runs rec-fun in parallel over asts
;; but bubbles get intersected as types (and then restored to bitypes)
;; and body*... gets to run over the types
(define-syntax parallel-intersected
  (syntax-rules ()
    [(_ parent-ast asts rec-fun reachable results idents)
     (begin
       (define idents
         (parallel*
          parent-ast #f #t type=?
          (lambda (node)
            (make-down-ref
             (make-AST-node
              parent-ast make-flow-graph-function-node
              "bitype->type" type=? bitype->type
              (up-ref-node node))))
          (lambda (node new) (type-intersect node (bitype->type new)))
          asts
          (parallel-loop asts asts rec-fun)))

       (define reachable
         ;; determine reachability for bottom vars (b/c might not
         ;; be bare) intersect variables that appear twice (stop if
```

```
          ;; any is bottom) (NOTE: singlely mentioned vars don't
          ;; need to be checked for bottom because they don't
          ;; intersect with each other.  If they were bottom then
          ;; the argument *value* would be bottom)
          (make-AST-for-all-non-bot-node
           parent-ast "reachable-by-env"

           ;; determine reachability from values
           (make-AST-for-all-non-bot-node
            parent-ast "reachable-by-value" (map* AST-output-node asts))

           (map* (lambda (ident) (down-ref-node (ident-preceding-ref ident)))
                 idents)))

        (define (intersected-node ast)
          (define var (AST-Ref->ident ast #f))
          ;; feed back if (1) it is a var and (2) it is an intersected
          ;; var, i.e., don't feed back singlely mentioned variables
          ;; because they will be up-refs not down-refs

          ;; NOTE: memq is O(1) b/c on a var the bubble length is at most one
          (if (and var (memq var (AST-bubble-vars ast)))
              (down-ref-node (ident-preceding-ref var))
              (AST-output-node ast)))

        ;; feed back bare vars to formal positions
        (define results (map* intersected-node asts))
        (define-ignored
          (for-each* (lambda (ident)
                       (turn-var-up! parent-ast "parallel" type->bitype ident))
                     idents)))])))

(define (move-var-to-here ast var bypass-node)
  (define ref (ident-preceding-ref var))
  (define preceding-ast (up-ref-ast ref))
  (define preceding-node (up-ref-node ref))
  (if (eq? ast preceding-ast) ;; may already be here b/c of parallel
      preceding-node
      (let* ([env-flow-node
              (composition-stack-range
               (make-AST-const-node ast "tf-id" env-flow-function-identity)
               (AST-graph-context preceding-ast)
               (AST-graph-context ast))]
             [new-output-node
              (make-AST-node
               ast make-flow-graph-function-node
               (string-append "tf-apply:" (symbol->string (ident-symbol-name var)))
               bitype=? env-flow-function-apply
               env-flow-node preceding-node bypass-node)])
        (ident-preceding-ref-set! var (make-up-ref ast new-output-node))
        new-output-node)))

(define (AST-bypass-nodes ast)
  (define ast-bubble-vars (AST-bubble-vars ast))
  (define bypass-ref (map* ident-preceding-ref ast-bubble-vars))
  (assert (for-all down-ref? bypass-ref))
  (map* down-ref-node bypass-ref))

(define (make-value-flow-graph ast)
```

```
  (define bot-node ;; Used for return of arg-count mismatch
    (make-AST-const-node ast "top-node" bot-type))
  (define bibot-node ;; Used for out-formal of arg-count mismatch
    (make-AST-const-node ast "top-node" (type->bitype bot-type)))
  (define bitop-node ;; Used for arguments beyond 'rest'
    (make-AST-const-node ast "bitop-node" (type->bitype top-type)))

  (define (make-const-proc type)
    (make-proc-record
     (lambda () '())
     (lambda (arg-count callable) '())
     (lambda (arg-count position arg-type)
       (when (eq? type top-type) (mark-escaped-type! arg-type))
       '())
     (let ([node (make-AST-const-node ast "surrogate-proc-return" type)])
       (lambda (arg-count) node))
     (let ([node (make-AST-const-node
                  ast "surrogate-proc-arg" (type->bitype type))])
       (lambda (arg-count position) node))))
  (define top-proc (make-const-proc top-type))
  (define bot-proc (make-const-proc bot-type))
  (define (type->proc-record* arg-count)
    (lambda (reachable type)
      (select-CFA
       [(zero-CFA)
        (if (not reachable)
            ;; list-of bot-proc here so that we get the correct return
            ;; type out of the surrogate-fun calls
            (list bot-proc)
            (let ([proc* (type-procs type)])
              (if (null? proc*)
                  (list bot-proc)
                  (begin
                    (for-each
                     (lambda (proc)
                       (unless (eq? proc #t)
                         ((proc-record-input-called-proc proc) arg-count #t)))
                     proc*)
                    (map (lambda (proc) (if (eq? proc #t) top-proc proc))
                         proc*)))))]
       [(sub-zero-CFA)
        (if (not reachable)
            bot-proc
            (proc-type-bits-case (type-procs type)
              [(bot-proc-type-bits) bot-proc]
              [(top-proc-type-bits) top-proc]
              [(one-or-more-proc-type-bits proc)
               ((proc-record-input-called-proc proc) arg-count #t)
               proc]))])))

  (define (rec-true ast reachable)
    (define rec rec-true)
    (define ast-bubble-vars (AST-bubble-vars ast))
    (define bypass-nodes (AST-bypass-nodes ast))

    (AST-output-node-set!
     ast
     (AST-case ast
```

```
        [(Ref var)
         (let ([ref (ident-preceding-ref var)])
           (assert (down-ref? ref))
           ;; update preceding-ref to be here
           (unless (ident-assigned var)
             (turn-var-up! ast "ref" type->split-bitype var))
           (make-AST-type-guard-node ast "ref" reachable (down-ref-node ref)))]

        [(Quote obj)
         (make-AST-const-type-guard-node ast "quote" reachable (type-of obj))]

        [(Primref safe name)
         (make-AST-const-type-guard-node
          ast "Primref" reachable (prim-name->type ast name))]

        [(Set ident body)
         (rec body reachable)
         (make-AST-binding-node-add! ast "set!-var" ident (AST-output-node body))
         (make-AST-const-type-guard-node
          ast "set!" (AST->bool-node ast "set!-reach" non-bot-type? body) top-type)]

        [(Seq expr1 expr2)
         (rec expr1 reachable)
         (for-each* (lambda (x) (turn-var-down! ast "seq" bitype->type x))
                    (AST-bubble-vars expr1))
         (rec expr2 (AST->bool-node ast "reach" non-bot-type? expr1))
         (AST->type-id-node ast "seq" expr2)]

        [(If test conseq altern)
         (rec test reachable)

         (parallel*
          ast #t #t bitype=? (lambda (x) x) bitype-union (list conseq altern)
          (parallel-list
           [conseq
            (begin
              ;; Turn those mentioned in test and conseq back down conseq
              (for-each*
               (lambda (x) (turn-var-down! ast "conseq" bitype-true x))
               (AST-bubbles-intersect (AST-bubble-vars test)
                                      (AST-bubble-vars conseq)))
              (rec conseq (AST->bool-node ast "reach-conseq" true-type? test)))]
           [altern
            (begin
              ;; Turn those mentioned in test and altern back down altern
              (for-each*
               (lambda (x) (turn-var-down! ast "altern" bitype-false x))
               (AST-bubbles-intersect (AST-bubble-vars test)
                                      (AST-bubble-vars altern)))
              (rec altern (AST->bool-node ast "reach-altern" false-type? test)))]))

         (make-AST-node
          ast make-flow-graph-function-node "union" type=? type-union
          (AST-output-node conseq) (AST-output-node altern))]

        [(Letrec bindings body)
         (let ([lhs (map* letrec-binding-lhs (vector->list bindings))]
               [rhs (map* letrec-binding-rhs (vector->list bindings))])
```

```
          ;; Set-up bindings
          (define-ignored
            (for-each*
             (lambda (var) (make-AST-var-binding-node ast "letrec-binding" var))
             lhs))

          ;; intersect rhs
          (parallel-intersected
           ast rhs (lambda (rhs) (rec rhs reachable))
           body-reachable intersected-rhs idents)

          ;; Wire output of rhs to lhs bindings
          (for-each*
           (lambda (var node)
             (make-AST-binding-node-add! ast "letrec-binding-add" var node))
           lhs intersected-rhs)

          ;; Do the body
          (for-each*
           (lambda (var) (turn-var-down! ast "letrec" bitype->type var))
           (AST-bubble-vars body))
          (rec body body-reachable)

          ;; remove vars bound by letrec
          (for-each* ident-preceding-ref-delete! lhs)

          ;; return
          (AST->type-id-node ast "letrec-out" body))]

        [(CaseLambda table clauses)
         (let () ;; needed to allow 'define'
           (define (make-out-formal clause var in-formal)
             (define ref (ident-preceding-ref var))
             (if (down-ref? ref)
                 ;; turn the var back up if it was never reffed
                 (let ([node (down-ref-node ref)])
                   (assert (eq? in-formal node))
                   (make-AST-node
                    ast make-flow-graph-function-node
                    "unreffed-or-set!-var" bitype=? type->bitype node))
                 ;; move all remaining vars to body result
                 (make-AST-bitype-id-node
                  clause (string-append "out-formal:" (symbol->string
                                                       (ident-symbol-name var)))
                  (move-var-to-here (case-lambda-clause-body clause)
                                    var in-formal))))

           (define (calculate-clause clause)
             (define formal-idents (case-lambda-clause-formals clause))
             (define rest-ident (case-lambda-clause-rest clause))
             (define formals&rest-ident (case-lambda-clause-formals&rest clause))

             ;;;; input nodes
             (define in-called (make-AST-enable-node clause "in-called"))
             ;; setup vars bound by clause (don't set ast b/c it's down-var)
             (define in-formals&rest
               (map*
                (lambda (x) (make-AST-var-binding-node clause "lambda-binding" x))
                formals&rest-ident))
```

```
                ;;;; internal nodes
                (define in-reachable
                  (make-AST-for-all-non-bot-node
                   clause "in-formals-reachable" in-called in-formals&rest))

                ;; do body
                (define-ignored (rec (case-lambda-clause-body clause) in-reachable))

                ;;;; output nodes
                (define out-return
                  (AST->type-id-node clause "case-lambda-return"
                                     (case-lambda-clause-body clause)))
                (define out-formals&rest
                  (map* (lambda (var in-formal) (make-out-formal clause var in-formal))
                        formals&rest-ident in-formals&rest))

                ;; remove formal vars bound by clause
                (define-ignored
                  (for-each* ident-preceding-ref-delete! formals&rest-ident))

                ;; Set the clause info
                (case-lambda-clause-info-set!
                 clause
                 (make-clause-info
                  clause rest-ident has-escaped in-called in-formals&rest
                  out-return out-formals&rest bitop-node)))

              (define has-escaped (make-AST-enable-node ast "has-escaped"))

              ;; We don't need to intersect the bubble vars (they are
              ;; always bypassed) but we must do a parallel to keep the up
              ;; vs down refs correct.

              (let* ([clauses (vector->list clauses)]
                     [asts (map* case-lambda-clause-body clauses)])
                (parallel* ast #t #t bitype=? #f #f asts
                           (parallel-loop
                            asts clauses
                            (lambda (clause)
                              (define body (case-lambda-clause-body clause))
                              (let ([bubble-vars (AST-bubble-vars body)]
                                    [bypass-nodes (AST-bypass-nodes body)])
                                (calculate-clause clause)
                                (map* (lambda (x y) (move-var-to-here ast x y))
                                      bubble-vars bypass-nodes))))))

              (make-AST-node
               ast make-flow-graph-function-node "mark-escapes" eq?
               (lambda (has-escaped)
                 (when has-escaped
                   (for-each*
                    (lambda (arg-count)
                      ((input-called-proc table clauses #f) arg-count #t)
                      (for-each*
                       (lambda (arg-position)
                         ((input-argument-proc table clauses #f)
                          arg-count arg-position top-type))
                       (iota arg-count)))
```

```
                (iota (vector-length table))))
730           #f)
             has-escaped)
            (vector-for-each
             (lambda (clause)
               (make-AST-node
735             ast make-flow-graph-function-node "out-return-mark-escapes" eq?
                (lambda (has-escaped out-return)
                  (when has-escaped (mark-escaped-type! out-return))
                  #f)
                has-escaped
740             (clause-info-out-return (case-lambda-clause-info clause))))
             clauses)

            (make-AST-const-type-guard-node
             ast "case-lambda" reachable
745          (make-proc-type
              (escape-proc has-escaped)
              (input-called-proc table clauses #f)
              (input-argument-proc table clauses #f)
              (output-return-proc table clauses bot-node)
750           (output-argument-proc table clauses bibot-node))))]

          [(App fun actuals)
           (let () ;; needed to allow 'define'
             (define fun&actuals (cons fun (vector->list actuals)))
755          (define arg-count (vector-length actuals))

             ;;;; Pre-call
             ;; intersect fun&actuals
             (parallel-intersected
760           ast fun&actuals (lambda (expr) (rec expr reachable))
              call-reachable intersected-fun&actuals idents)

             ;; NOTE: also signals fun as callable
             (define surrogate-fun
765            (make-AST-node
                ast make-flow-graph-function-node "surrogate-fun" proc-type-bits=?
                (type->proc-record* arg-count)
                call-reachable (car intersected-fun&actuals)))
             (define intersected-actuals (cdr intersected-fun&actuals))
770
             ;;;; Call entry

             ;; send arguments to function
             (define-ignored
775            (for-each*
                (lambda (position intersected-actual)
                  (make-AST-node
                   ast make-flow-graph-function-node "input-argument" eq?
                   (select-CFA
780                 [(zero-CFA)
                    (lambda (proc-type* intersected-actual)
                      (for-each (lambda (proc-type)
                                  ((proc-record-input-argument-proc proc-type)
                                   arg-count position intersected-actual))
785                             proc-type*))]
                    [(sub-zero-CFA)
                     (lambda (proc-type intersected-actual)
```

```
                ((proc-record-input-argument-proc proc-type)
                 arg-count position intersected-actual))])
790           surrogate-fun intersected-actual))
      (iota arg-count)
      intersected-actuals))

;;;; Call return
(define app-out-return
  (make-AST-node
   ast make-flow-graph-join-node "app-return" type=?
   (select-CFA
    [(zero-CFA)
     (lambda (proc-type*)
       (when (null? proc-type*)
         (error 'app-out-return "expected␣at␣least␣one␣proc-type"))
       (fold-left
        (lambda (node proc-type)
          (make-AST-node
           ast make-flow-graph-function-node "return-value" type=?
           (lambda (type1 type2) (type-union type1 type2))
           node ((proc-record-output-return-proc proc-type) arg-count)))
        ((proc-record-output-return-proc (car proc-type*)) arg-count)
        (cdr proc-type*)))]
    [(sub-zero-CFA)
     (lambda (proc-type)
       ((proc-record-output-return-proc proc-type) arg-count))])
   surrogate-fun))

(define app-out-formals
  (map*
   (lambda (position)
     (make-AST-node
      ast make-flow-graph-join-node "app-out-formal" bitype=?
      (select-CFA
       [(zero-CFA)
        (lambda (proc-type*)
          (when (null? proc-type*)
            (error 'app-out-return "expected␣at␣least␣one␣proc-type"))
          (fold-left
           (lambda (node proc-type)
             (make-AST-node
              ast make-flow-graph-function-node "arg-out-value" bitype=?
              (lambda (bitype1 bitype2) (bitype-union bitype1 bitype2))
              node ((proc-record-output-argument-proc proc-type)
                    arg-count position)))
           ((proc-record-output-argument-proc (car proc-type*))
            arg-count position)
           (cdr proc-type*)))]
       [(sub-zero-CFA)
        (lambda (proc-type)
          ((proc-record-output-argument-proc proc-type)
           arg-count position))])
      surrogate-fun))
   (iota arg-count)))

;;;; Post-call

;; intersect formal output vars
(define-ignored
```

```
  (for-each*
   (lambda (position actual formal)
     (let ([var (AST-Ref->ident actual #f)])
       (when (and var (not (ident-assigned var)))
         (ident-preceding-ref-set!
          var
          (make-up-ref
           ast
           (make-AST-node
            ast make-flow-graph-function-node
            "post-call-var-intersect" bitype=?
            ;; NOTE: memq is O(1) b/c on a var the bubble length
            ;; is at most one
            (if (memq var (AST-bubble-vars actual))
                ;; The var is mentioned multiple times thus the
                ;; bitype has already been "void scrambled" by
                ;; the parallel-intersect so we don't have to do
                ;; that, but we new intersects need to be
                ;; true/false specific.  We shouldn't do
                ;; it because we might not be the first mention
                ;; and we don't want to scramble info from
                ;; other arguments.
                bitype-intersect
                ;; The var isn't mentioned multiple times so we
                ;; have to do our own scrambling.  This is OK,
                ;; since we are the only reference in this
                ;; application.
                (lambda (formal preceding)
                  (bitype-intersect
                   formal (bitype->void-bitype preceding))))
            formal (up-ref-node (ident-preceding-ref var)))))))))
   (iota (vector-length actuals))
   (vector->list actuals)
   app-out-formals))

;; determine post call return value
(define (app-out-reachable-by-value base actual formal)
  (make-AST-node
   ast make-flow-graph-function-node "app-out-reachable-by-value" type=?
   (lambda (base actual formal)
     (type-filter
      base
      (non-bot-type? (type-intersect actual (bitype-true formal)))
      (non-bot-type? (type-intersect actual (bitype-false formal)))))
   base actual formal))

(define (app-out-reachable-by-env base ident)
  (make-AST-node
   ast make-flow-graph-function-node "app-out-reachable-by-env" type=?
   (lambda (base bitype)
     (type-filter
      base
      (non-bot-type? (bitype-true bitype))
      (non-bot-type? (bitype-false bitype))))
   base
   (up-ref-node (ident-preceding-ref ident))))

(fold-left*
 app-out-reachable-by-env
```

```
            (fold-left* app-out-reachable-by-value
                        app-out-return intersected-actuals app-out-formals)
            idents))]))

    ;; compute new nodes based on previous reference and bypass
    (for-each* (lambda (x y) (move-var-to-here ast x y))
               ast-bubble-vars bypass-nodes))

  (rec-true ast (make-AST-const-node ast "<top>" #t)))

)
```

B.2.3. Post-processing

Listing B.12: ast-unparse.ss

```
(library (ast-unparse)
 (export AST->Trex)
 (import (rnrs) (srfi-39) (flow-graphs) (ast) (types) (value-flow-graphs)
         (prefix (tprogram) Trex:))

;; prim type table -- duplicates knowledge in prims
(define mark-safe?
  (let () ;; deeded to allow define
    (define (arity-match? arity actual-types)
      (if (>= arity 0)
          (= arity (length actual-types))
          (>= (+ (- arity) 1) (length actual-types))))
    (define (type-match? type actual-type)
      (let ([type (maybe-non-proc-type->type type)])
        (or (type=? type top-type) (type<=? actual-type type))))
    (define (type-with-rest-match? types actual-types)
      (let f ([types types] [actual-types actual-types])
        (if (or (null? actual-types) (null? types))
            #t
            (let ([type (car types)] [remaining-types (cdr types)])
              (and
               (type<=? (car actual-types) (maybe-non-proc-type->type type))
               (if (null? remaining-types)
                   (f types (cdr actual-types))
                   (f remaining-types (cdr actual-types)))))))))
    (define (types-match? sig actual-types)
      (let ([arity (car sig)] [types (cdr sig)])
        (if (>= arity 0)
            (for-all type-match? types actual-types)
            (type-with-rest-match? types actual-types))))
    (define-syntax build-prim-entry
      (syntax-rules ()
        [(_ ht (prim* ...) ?value)
         (let ([value ?value])
           (hashtable-set! ht 'prim* value) ...)]
        [(_ ht prim value) (hashtable-set! ht 'prim value)]))
    (define-syntax define-prim-table
      (lambda (x)
```

```
      (define (ellipsis? x)
        (and (identifier? x) (free-identifier=? x #'(... ...))))
      (define (find-arity arg*)
        (syntax-case arg* ()
          [(arg arg* ... dots)
           (ellipsis? #'dots)
           #`(#,(fx- -1 (length #'(arg arg* ...))) arg arg* ...)]
          [(arg* ...) #`(#,(length #'(arg* ...)) arg* ...)]))
      (syntax-case x (sig)
        [(_ name (name* [sig arg** ...] extras** ...) ...)
         (with-syntax ([(((arity** arg*** ...) ...) ...)
                        (map (lambda (arg**)
                               (map find-arity arg**))
                             #'((arg** ...) ...))])
           #'(define name
               (let ([ht (make-eq-hashtable)])
                 (build-prim-entry
                  ht name* (list (list (list arity** arg*** ...) ...)
                                 (list extras** ...))) ...
                 ht)))]))))
  (define-prim-table prim-table
    ((car cdr) [sig (pair-type-bits)])
    ((set-car! set-cdr!) [sig (pair-type-bits top-type)])
    (make-vector [sig (length-type-bits) (length-type-bits top-type)])
    (vector-length [sig (vector-type-bits)])
    (vector-ref [sig (vector-type-bits length-type-bits)]
      (let ([vector-type (maybe-non-proc-type->type vector-type-bits)]
            [length-type (maybe-non-proc-type->type length-type-bits)])
        (lambda (src name types)
          (if (and (= (length types) 2) (type<=? (car types) vector-type))
              (if (type<=? (cadr types) length-type)
                  (Trex:Primref src #t '$vector-check-range-ref)
                  (Trex:Primref src #t '$vector-check-fx-range-ref))
              (Trex:Primref src #t name)))))
    (vector-set! [sig (vector-type-bits length-type-bits top-type)]
      (let ([vector-type (maybe-non-proc-type->type vector-type-bits)]
            [length-type (maybe-non-proc-type->type length-type-bits)])
        (lambda (src name types)
          (if (and (= (length types) 3) (type<=? (car types) vector-type))
              (if (type<=? (cadr types) length-type)
                  (Trex:Primref src #t '$vector-check-range-set!)
                  (Trex:Primref src #t '$vector-check-fx-range-set!))
              (Trex:Primref src #t name)))))
    ((fldiv flmod fldiv0 flmod0 flexpt) [sig (flonum-type-bits flonum-type-bits)])
    ((flatan fllog) [sig (flonum-type-bits) (flonum-type-bits flonum-type-bits)])
    ((flsqrt flabs flnumerator fldenominator flfloor flceiling fltruncate
      flround flexp flsin flcos fltan flasin flacos)
     [sig (flonum-type-bits)])
    ((fl+ fl*) [sig (flonum-type-bits ...)])
    ((fl- fl/ flmax flmin)
     [sig (flonum-type-bits) (flonum-type-bits flonum-type-bits ...)])
    ((flpositive? flnegative? flzero? flinteger?
      flodd? fleven?  flfinite? flinfinite? flnan?)
     [sig (flonum-type-bits)])
    ((fl>? fl<? fl>=? fl<=? fl=?)
     [sig (flonum-type-bits flonum-type-bits flonum-type-bits ...)])
    (fxxor [sig (fxnum-type-bits ...)])
    ((fx+ fx*) [sig (fxnum-type-bits ...)])
    ((fx- fx/) [sig (fxnum-type-bits fxnum-type-bits ...)])
```

```
      ((fxarithmetic-shift-left fxarithmetic-shift-right fxarithmetic-shift)
       [sig (fxnum-type-bits fxnum-type-bits)])
      ((fx=? fx<=? fx>=? fx<? fx>?)
       [sig (fxnum-type-bits fxnum-type-bits fxnum-type-bits ...)])
      ((fx<= fx< fx= fx>= fx>) [sig (fxnum-type-bits fxnum-type-bits ...)])
      ((char=? char<=? char<? char>? char>=?)
       [sig (char-type-bits char-type-bits ...)])
      ((r6rs:char=? r6rs:char<=? r6rs:char<? r6rs:char>? r6rs:char>=?)
       [sig (char-type-bits char-type-bits char-type-bits ...)])
      (integer->char [sig (fxnum-type-bits)])
      (char->integer [sig (char-type-bits)])
      (string-length [sig (string-type-bits)])
      (string [sig (char-type-bits ...)])
      (make-string [sig (fxnum-type-bits) (fxnum-type-bits char-type-bits)])
      ;; can do better with this, need to handle like vector-set! and vector-ref
      (string-set! [sig (string-type-bits fxnum-type-bits char-type-bits)])
      (string-ref [sig (string-type-bits fxnum-type-bits)]))
    (lambda (src prim actual-types)
      (define (check-sig sig)
        (and (arity-match? (car sig) actual-types)
             (types-match? sig actual-types)))
      (define (check-prim name sigs extras)
          (cond
           [(and (pair? extras) (procedure? (car extras)))
            ((car extras) src name actual-types)]
           [(not (null? extras)) #f]
           [(exists check-sig sigs) (Trex:Primref src #f name)]
           [else #f]))
      (cond
       [(hashtable-ref prim-table prim #f) =>
        (lambda (v) (apply check-prim prim v))]
       [else #f]))))

(define (map-values vals f ls . more)
    (define (return-values vals vals*)
      (if (null? vals*)
          (apply values (map list vals))
          (apply values (map cons vals vals*))))
    (if (null? more)
        (let map1 ([ls ls])
          (if (null? ls)
              (apply values vals)
              (let-values ([vals (f (car ls))] [vals* (map1 (cdr ls))])
                (return-values vals vals*))))
        (let map-more ([ls ls] [more more])
          (if (null? ls)
              (apply values vals)
              (let-values ([vals (apply f (car ls) (map car more))]
                           [vals* (map-more (cdr ls) (map cdr more))])
                (return-values vals vals*))))))

(define (AST-output-type ast) (flow-graph-node-output (AST-output-node ast)))

(define (AST->Trex ast)
  (define changed (make-parameter #f))

  (define (rec ast)
    (define src (AST-src ast))
    (values
```

```
(AST-case ast
  ;; Things that don't collapse
  [(Ref id) (Trex:Ref src id)]
  [(Quote obj) (Trex:Quote src obj)]
  [(Primref safe name) (Trex:Primref src safe name)]
  [(Set id body) (let-values ([(body body-type) (rec body)])
                   (Trex:Set src id body))]
  [(App fun actuals)
   (let-values ([(actuals actual-types)
                 (map-values (list '() '()) rec (vector->list actuals))])
     (AST-case fun
       [(Primref safe name)
        (cond
         [(and safe (mark-safe? (AST-src fun) name actual-types)) =>
          (lambda (fun) (Trex:App src fun actuals))]
         [else
          (let-values ([(fun fun-type) (rec fun)])
            (Trex:App src fun actuals))])]
       [else
        (let-values ([(fun fun-type) (rec fun)])
          (Trex:App src fun actuals))]))]
  [(Letrec bindings body)
   (let ([bindings (vector->list bindings)])
     (let-values ([(body type) (rec body)]
                  [(rhs rhs-types)
                   (map-values (list '() '()) rec
                               (map letrec-binding-rhs bindings))])
       (Trex:Letrec src (map letrec-binding-lhs bindings) rhs body)))]
  [(CaseLambda table clauses)
   (let ([Clause->Trex
          (lambda (clause)
            (let-values ([(body type) (rec (case-lambda-clause-body clause))])
              (Trex:CaseLambdaClause
               (append (vector->list (case-lambda-clause-formals clause))
                       (or (case-lambda-clause-rest clause) '()))
               body)))]
         [Clause-called
          (lambda (clause)
            (flow-graph-node-output
             (clause-info-in-called (case-lambda-clause-info clause))))])
     (Trex:CaseLambda
      src (map Clause->Trex
               (filter Clause-called (vector->list clauses)))))]

  ;; Things that might collapse
  [(If test conseq altern)
   (let-values ([(test type) (rec test)]
                [(conseq c-type) (rec conseq)]
                [(altern a-type) (rec altern)])
     (let ([conseq-reachable (true-type? type)]
           [altern-reachable (false-type? type)])
       (cond
        [(and conseq-reachable altern-reachable)
         (Trex:If src test conseq altern)]
        [conseq-reachable (changed #t) (Trex:Seq src test conseq)]
        [altern-reachable (changed #t) (Trex:Seq src test altern)]
        [else (changed #t) test])))]
  [(Seq expr1 expr2)
   (let-values ([(expr1 e1-type) (rec expr1)]
```

```
                    [(expr2 e2-type) (rec expr2)])
          (if (non-bot-type? e1-type)
              (Trex:Seq src expr1 expr2)
              (begin (changed #t) expr1))])
     (AST-output-type ast)))

  (let ([trex
         (AST-case ast
           [(App fun actuals)
            (AST-case fun
              [(Primref safe name) ;; assumes "escape" is always unsafe
               (cond
                [(eq? name 'escape)
                 (cond
                  [(= 1 (vector-length actuals))
                   (let-values ([(body type) (rec (vector-ref actuals 0))])
                     body)]
                  [else #f])]
                [else #f])]
              [else #f])])])
    (or trex (assertion-violation
              'AST->Trex "Incorrect toplevel program expression" ast))))

)
```

B.3. Data types

B.3.1. Abstract syntax trees

Listing B.13: idents.ss

```
(library (idents)
  (export
   make-ident ident? ident-name ident-symbol-name
   ident-number ident-number-set!
   ident-assigned ident-assigned-set!
   ident-locations ident-locations-set!
   ident-asts ident-asts-set!
   ident-preceding-ref ident-preceding-ref-set! ident-preceding-ref-delete!
   ident-binding-node ident-binding-node-set!
   ident=?
   ident<?
   ident>?
   )
  (import (rnrs) (records-io))

(define-record-type ident
  (fields name symbol-name
          (mutable assigned) ;; bool
          (mutable number) ;; int
```

```
          (mutable asts) ;; list[ast(ref|set)]
          (mutable locations) ;; list[int]
          (mutable preceding-ref) ;; up-ref or down-ref
          (mutable binding-node) ;; node[type]
          )
  (nongenerative)
  (protocol
   (lambda (make)
     (lambda (name symbol-name assigned)
       (make name symbol-name assigned -1 '() '() #f #f)))))

(define (ident=? x y) (= (ident-number x) (ident-number y)))
(define (ident<? x y) (< (ident-number x) (ident-number y)))
(define (ident>? x y) (> (ident-number x) (ident-number y)))
(define (ident-preceding-ref-delete! ident)
  (ident-preceding-ref-set! ident #f))

(record-writer (record-type-descriptor ident)
  (lambda (r p wr)
    (display "#[ident␣" p)
    (display (ident-symbol-name r) p)
    (display "␣" p)
    (write (ident-assigned r) p)
    (display "]" p)))

)
```

Listing B.14: tprogram.ss

```
(library (tprogram)
  (export Trex? Trex-case
          Ref Quote Primref PrimrefWrap Set App If CaseLambda Seq
          Letrec Trex-CaseLambdaClause-case CaseLambdaClause)
  (import (rnrs) (datatype) (idents))

(define-datatype (Trex-CaseLambdaClause)
  (CaseLambdaClause (formals (improper-list-of ident?)) (body Trex?)))

(define-datatype (Trex)
  (Ref src (id ident?))
  (Quote src obj)
  (Primref src (safe boolean?) prim-identifier)
  (PrimrefWrap (primref Trex?) (expr Trex?)) ;; Used by benchmarking code
  (Set src (id ident?) (expr Trex?))
  (App src (expr Trex?) (expr* (list-of Trex?)))
  (If src (test Trex?) (thn Trex?) (els Trex?))
  (CaseLambda src (clause* (list-of Trex-CaseLambdaClause?)))
  (Seq src (expr1 Trex?) (expr2 Trex?))
  (Letrec src (id* (list-of ident?)) (expr* (list-of Trex?)) (body Trex?)))

)
```

Listing B.15: ast.ss

```
(library (ast)
```

```
  (export
   AST? AST-case Ref Quote Primref Set App If CaseLambda Seq Letrec

   Trex->AST

   AST-src ;; obj
   AST-context ;; myers-stack of AST
   AST-output-node ;; flow-graph-node of type
   AST-output-node-set!
   AST-graph-context ;; composition-stack of env-flow-functions
   AST-graph-context-set!
   AST-graph-context-thunk ;; (lambda () ...)
   AST-graph-context-thunk-set!
   AST-contained-nodes ;; list of flow-graph-node
   AST-contained-nodes-set!
   AST-bubble-vars ;; list of ident
   AST-bubble-vars-set!
   AST-tree-start-location ;; int
   AST-tree-start-location-set!
   AST-tree-end-location ;; int
   AST-tree-end-location-set!

   case-lambda-clause?
   case-lambda-clause-ref
   case-lambda-clause-formals case-lambda-clause-rest
   case-lambda-clause-formals&rest case-lambda-clause-body
   case-lambda-clause-info case-lambda-clause-info-set!
   case-lambda-clause-contained-nodes case-lambda-clause-contained-nodes-set!

   letrec-binding-lhs letrec-binding-rhs
   )

  (import (rnrs) (prefix (tprogram) Trex:)
          (idents) (srfi-39) (datatype) (myers-stacks)
          (only (composition-stacks) composition-stack?)
          (only (flow-graphs) flow-graph-node?))

(define-datatype
  (AST (src #f)
       (output-node #f flow-graph-node?)
       (contained-nodes '() #;(list-of flow-graph-node?))
       (context #f myers-stack?) ;; myers stack of ast-nodes including self
       (graph-context #f composition-stack?)
       (graph-context-thunk #f #;(pair flow-graph-node? (list-of ast)))
       (bubble-vars '() #;(list-of ident?))
       (tree-start-location #f integer?)
       (tree-end-location #f integer?)
       )

  (Ref (id ident?))
  (Quote obj)
  (Primref (safe boolean?) (name symbol?))
  (Set (id ident?) (expr AST?))
  (App (expr AST?) (expr* (vector-of AST?)))
  (If (test AST?) (thn AST?) (els AST?))
  (CaseLambda (table (vector-of (lambda (x) (or (not x) (integer? x)))))
              (clause* (vector-of case-lambda-clause?)))
  (Seq (expr1 AST?) (expr2 AST?))
  (Letrec (binding* (vector-of letrec-binding?)) (body AST?)))
```

```
(define-record-type letrec-binding
  (nongenerative)
  (fields lhs rhs)
  (protocol
   (lambda (make-letrec-binding)
     (lambda (lhs rhs)
       (assert (ident? lhs))
       (assert (AST? rhs))
       (make-letrec-binding lhs rhs)))))

(define-record-type case-lambda-clause
  (nongenerative)
  (fields formals rest body formals&rest
          (mutable info) (mutable contained-nodes))
  (protocol
   (lambda (make)
     (lambda (formals rest body)
       (assert ((vector-of ident?) formals))
       (assert (or (not rest) (ident? rest)))
       (assert (AST? body))
       (make formals rest body
             (append (vector->list formals) (if rest (list rest) '()) #f '()))))))

;; vector[case-lambda-clause] -> vector[#f or clause-index]
;;   last slot is for arguments beyond vector-length
(define (make-case-lambda-table clauses)
  (define (vector-set-or! vector index value)
    (vector-set! vector index (or (vector-ref vector index) value)))
  (define has-rest (vector-map case-lambda-clause-rest clauses))
  (define num-non-rest-args
    (vector-map vector-length
                (vector-map case-lambda-clause-formals clauses)))
  (define table-length (+ 2 (apply max (cons 0 (vector->list num-non-rest-args)))))
  (define table (make-vector table-length #f))
  (let clause-loop ([clause-number 0][min-rest +inf.0])
    (unless (= clause-number (vector-length clauses))
      (let ([table-index (vector-ref num-non-rest-args clause-number)])
        (if (not (vector-ref has-rest clause-number))
            (begin
              (vector-set-or! table table-index clause-number)
              (clause-loop (+ 1 clause-number) min-rest))
            (begin
              (let rest-loop ([table-index table-index])
                (unless (or (>= table-index min-rest)
                            (>= table-index table-length))
                  (vector-set-or! table table-index clause-number)
                  (rest-loop (+ 1 table-index))))
              (clause-loop (+ 1 clause-number) (min min-rest table-index)))))))
  table)

;; case-lambda-table * vector[case-lambda-clause] * int -> #f or case-lambda-clause
(define (case-lambda-clause-ref table clauses arg-count)
   (let ([index (vector-ref table (min arg-count (- (vector-length table) 1)))])
     (and index (vector-ref clauses index))))

(define (Trex->AST trex)
  (define (source src ast) (AST-src-set! ast src) ast)
```

```
(define (rec0 trex)
  (define (rec trex) (rec0 trex))

  (define (CaseLambdaClause->AST clause)
    (define (improper-list-split list default)
      (cond
       [(null? list) (values '() default)]
       [(pair? list)
        (let-values ([(front rest) (improper-list-split (cdr list) default)])
          (values (cons (car list) front) rest))]
       [else (values '() list)]))
    (Trex:Trex-CaseLambdaClause-case clause
      [(Trex:CaseLambdaClause formals body)
       (let-values ([(front rest) (improper-list-split formals #f)])
         (make-case-lambda-clause (list->vector front) rest (rec body)))]))

  (Trex:Trex-case trex
    [(Trex:Ref src id) (source src (Ref id))]
    [(Trex:Quote src obj) (source src (Quote obj))]
    [(Trex:Primref src safe name) (source src (Primref safe name))]
    [(Trex:Set src id expr) (source src (Set id (rec expr)))]
    [(Trex:App src expr expr*)
     (source src (App (rec expr) (list->vector (map rec expr*))))]
    [(Trex:If src test thn els)
     (source src (If (rec test) (rec thn) (rec els)))]
    [(Trex:CaseLambda src clause*)
     (let ([clause* (list->vector (map CaseLambdaClause->AST clause*))])
       (source src (CaseLambda (make-case-lambda-table clause*) clause*)))]
    [(Trex:Seq src expr1 expr2)
     (source src (Seq (rec expr1) (rec expr2)))]
    [(Trex:Letrec src lhs rhs body)
     (source src (Letrec
                  (list->vector (map make-letrec-binding lhs (map rec rhs)))
                  (rec body)))]))

(define (rec1 ast context)
  (let ([context (myers-stack-cons ast context)])
    (define (rec ast) (rec1 ast context))
    (AST-context-set! ast context)

    (AST-case ast
      [(Ref var) (values)]
      [(Quote obj) (values)]
      [(Primref safe name) (values)]
      [(Set var expr) (rec expr)]
      [(App fun actuals) (rec fun) (vector-for-each rec actuals)]
      [(If test conseq altern) (rec test) (rec conseq) (rec altern)]
      [(CaseLambda table clauses)
       (vector-for-each (lambda (clause) (rec (case-lambda-clause-body clause)))
                        clauses)]
      [(Seq expr1 expr2) (rec expr1) (rec expr2)]

     [(Letrec bindings body)
      (vector-for-each (lambda (binding) (rec (letrec-binding-rhs binding)))
                       bindings)
      (rec body)])))

(define ast (rec0 trex))
(rec1 ast myers-stack-null)
```

```
  ast)

)
```

B.3.2. Flow graphs

Listing B.16: flow-graphs.ss

```
(library (flow-graphs)
  (export

   make-flow-graph ;; void -> flow-graph
   flow-graph? ;; obj -> bool
   flow-graph-step! ;; flow-graph -> void
   flow-graph-stable? ;; flow-graph -> bool
   flow-graph-print
   ;; (printer :
   ;;   (contain : string * list of flow-graph-node * list of obj -> void) *
   ;;   (node : obj) -> void) *
   ;; (labeler : flow-graph-node -> string) *
   ;; obj -> void

   flow-graph-node? ;; obj -> bool
   flow-graph-node-output ;; flow-graph-node -> obj
   flow-graph-node-info ;; flow-graph-node -> obj

   ;; flow-graph * info * equ? * output -> flow-graph-input-node
   make-flow-graph-mutable-node

   ;; flow-graph-mutable-node? ;; obj -> bool
   flow-graph-mutable-node-set! ;; flow-graph-input-node * output -> void

   ;; flow-graph * info * equ? * function * output -> flow-graph-lazy-node
   make-flow-graph-lazy-node

   ;; flow-graph-lazy-node? ;; obj -> bool
   flow-graph-lazy-node-force! ;; flow-graph-lazy-node * args ... -> void

   ;; flow-graph * info * equ? * function * args ... -> flow-graph-function-node
   make-flow-graph-function-node

   ;; flow-graph-function-node? ;; obj -> bool

   ;; flow-graph * info * equ? * input -> flow-graph-join-node
   make-flow-graph-join-node

   ;;flow-graph-join-node? ;; obj -> bool
   )
  (import (except (rnrs) case cond if) (safe-forms) (records-io)
          (rnrs mutable-pairs))

;;;;;;;;;;;;;;;;
;; Flow Graphs
;;;;;;;;;;;;;;;;
```

```
(define-record-type flow-graph
  (nongenerative)
  (fields (mutable pending-list)) ;; TODO: as a rotating vector of lists
  (protocol (lambda (make) (lambda () (make '())))))

(define (flow-graph-pending-add! flow-graph node)
  (flow-graph-pending-list-set!
   flow-graph (cons node (flow-graph-pending-list flow-graph))))

(define (flow-graph-pending-remove! flow-graph)
  (let ([x (flow-graph-pending-list flow-graph)])
    (flow-graph-pending-list-set! flow-graph (cdr x))
    (car x)))

(define (flow-graph-step! graph)
  (define node (flow-graph-pending-remove! graph))
  (when (flow-graph-node-pending node)
        (flow-graph-node-pending-set! node #f)
        (node-output-set! node ((flow-graph-node-action node) node))))

(define (flow-graph-stable? graph) (null? (flow-graph-pending-list graph)))

;;;;;;;;;;;;;;;;
;; Graph Nodes
;;;;;;;;;;;;;;;;
(define-record-type flow-graph-node
  (nongenerative)
  (fields graph
          info
          (mutable eqv?)
          (mutable pending) ;; #t or #f
          (mutable output)
          #;(mutable outputs) ;; for debugging
          (mutable action) ;; void -> output
          (mutable listeners) ;; list of (node or (cons (node or #f) '()))
          )
  (protocol
   (lambda (make)
     (lambda (graph info eqv? output action)
       (assert (string? info))
       (assert (or (not eqv?) (eqv? output output))) ;; Sanity check
       (make graph info eqv? #f output action '())))))

;;;;;;;;;;;;;;;;
;; Lazy Node
;;;;;;;;;;;;;;;;
(define (flow-graph-lazy-node-action node)
  (error 'flow-graph-lazy-node "evaluating␣unforced␣lazy␣node" node))
(define (make-flow-graph-lazy-node graph info eqv? output)
  (make-flow-graph-node graph info eqv? output flow-graph-lazy-node-action))

(define-syntax flow-graph-lazy-node-force!
  (lambda (stx)
    (syntax-case stx ()
      [(_ n fun args ...)
       (with-syntax ([(arguments ...) (generate-temporaries #'(args ...))])
         #'(let ([node n]
                 [function fun]
```

```
                [arguments args] ...)
              (assert (eq? (flow-graph-node-graph node)
                           (flow-graph-node-graph arguments))) ...
              (assert (eq? (flow-graph-node-action node)
                           flow-graph-lazy-node-action))
              (flow-graph-node-action-set!
               node (lambda (node)
                      (function (flow-graph-node-output arguments) ...)))
              (add-to-listeners! arguments node) ...
              (add-to-pending! node))]))))

;;;;;;;;;;;;;;;;
;; Function Node
;;;;;;;;;;;;;;;;
(define-syntax make-flow-graph-function-node
  (lambda (stx)
    (syntax-case stx ()
      [(_ graph info eqv? function arguments ...)
       (with-syntax ([(fun args ...)
                      (generate-temporaries #'(function arguments ...))])
         #'(let ([g graph]
                 [fun function]
                 [args arguments] ...)
             (assert (eq? g (flow-graph-node-graph args))) ...
             (let ([action (lambda (node) (fun (flow-graph-node-output args) ...))])
               (let ([node (make-flow-graph-node g info eqv? (action #f) action)])
                 (add-to-listeners! args node) ...
                 node))))])))

;;;;;;;;;;;;;;;;
;; Mutable Node
;;;;;;;;;;;;;;;;
(define (flow-graph-mutable-node-action node)
  (error 'flow-graph-mutable-node "shouldn't be on pending queue" node))
(define (make-flow-graph-mutable-node graph info eqv? output)
  (make-flow-graph-node graph info eqv? output flow-graph-mutable-node-action))
(define (flow-graph-mutable-node? node)
  (and (flow-graph-node? node)
       (eq? (flow-graph-node-action node) flow-graph-mutable-node-action)))

(define (flow-graph-mutable-node-set! node new-output)
  (assert (eq? (flow-graph-node-action node) flow-graph-mutable-node-action))
  (node-output-set! node new-output))

;;;;;;;;;;;;;;;;
;; Join Node
;;;;;;;;;;;;;;;;
(define-syntax make-flow-graph-join-node
  (lambda (stx)
    (syntax-case stx ()
      [(_ graph info eqv? function arguments ...)
       (with-syntax ([(fun args ...)
                      (generate-temporaries #'(function arguments ...))])
         #'(let ([g graph]
                 [fun function]
                 [args arguments] ...)
             (assert (eq? g (flow-graph-node-graph args))) ...
             (let ([inner-node #f] ;; dummy value
                   [key (cons #f '())]) ;; dummy value
```

```
               (let ([action
                      (lambda (node)
                        (let ([new-inner (fun (flow-graph-node-output args) ...)])
                          ;; (assert (eq? graph (flow-graph-node-graph inner-node)))
                          (when (not (eq? inner-node new-inner))
                            (remove-from-join-listeners! key)
                            (set! inner-node new-inner)
                            (set! key (add-to-join-listeners! new-inner node)))
                          (flow-graph-node-output inner-node)))])
                 (let ([node (make-flow-graph-node graph info
                                                   #f #f ;; dummy values
                                                   action)])
                   (add-to-listeners! args node) ...
                   (let ([output (action node)])
                     (assert (eqv? output output)) ;; ensure dummy values skipped
                     (flow-graph-node-eqv?-set! node eqv?) ;; fix dummy value
                     (flow-graph-node-output-set! node output)) ;; fix dummy value
                   node)))))]))))

;;;;;;;;;;;;;;;;
;; Printing
;;;;;;;;;;;;;;;;
;; (printer :
;;   (contain : string * list of flow-graph-node * list of obj -> void) *
;;   (node : obj) -> void) *
;; (labeler : flow-graph-node -> string) *
;; obj -> void

(define (flow-graph-print container node-label node)
  ;; for indentation
  (define depth 0)
  (define (indent)
    (let loop ([i depth]) (unless (zero? i) (display "␣␣") (loop (- i 1)))))

  ;; for labeling nodes
  (define node-table (make-eq-hashtable))
  (define (node-table-add-node! node)
    (assert (not (hashtable-contains? node-table node)))
    (hashtable-set! node-table node (cons containers (hashtable-size node-table))))
  (define (node-id node)
    (cdr (or (hashtable-ref node-table node #f)
             (assertion-violation 'node-id "node␣doesn't␣exist" node))))
  (define (node-cluster node)
    (car (or (hashtable-ref node-table node #f)
             (assertion-violation 'node-container "node␣doesn't␣exist" node))))
  (define (node-name node) (string-append "node" (number->string (node-id node))))

  (define containers 0) ;; for labeling clusters

  ;; for printing missing edges
  (define node-container-table (make-eq-hashtable))

  (define (escape-label x) (string-append "\"" x "\""))
  (define edges '())

  (define (print-containment label nodes children)
    (indent)(display "subgraph␣cluster")(display containers)(display "␣{\n")
    (set! depth (+ depth 1))
    (set! containers (+ containers 1))
```

```
    (indent)(display "label␣=␣")(display (escape-label label))(display ";\n")
    (for-each (lambda (child) (container print-containment child)) children)
    (for-each (lambda (x) (print-node x #t)) nodes)
    (set! depth (- depth 1))
    (indent)(display "}\n"))

  (define (print-node node first-pass)
    (assert (flow-graph-node? node))
    (node-table-add-node! node)
    (when (and first-pass #t #;(flow-graph-lazy-node? node))
      (set! edges
            (append (map (lambda (x) (cons node x))
                         (filter (lambda (x) (not (pair? x)))
                                 (flow-graph-node-listeners node)))
                    edges)))
    (indent)
    (unless first-pass
      (display "subgraph␣cluster")(display (node-cluster node))(display "{␣"))
    (display (node-name node))
    (display "␣[")
    (display "label␣=␣")(display (escape-label (node-label node)))
    (display ",␣shape␣=␣")
    (display "ellipse")
    #;(cond
        [(flow-graph-mutable-node? node) (display "diamond")]
        [(flow-graph-lazy-node? node) (display "ellipse")])
    (when (flow-graph-node-pending node)
      (display ",␣penwidth␣=␣4"))
    (unless first-pass (display ",␣style=dotted"))
    (display "];")
    (unless first-pass (display "}"))
    (display "\n"))

  (display "digraph␣graph0␣{\n")
  (container print-containment node)
  (for-each (lambda (edge)
              (define from (car edge))
              (define to (cdr edge))
              (unless (hashtable-contains? node-table from) (print-node from #f))
              (display (node-name from))(display "␣->␣")
              (display (node-name to))(display ";\n"))
            edges)
  (display "}\n"))

;;;;;;;;;;;;;;;;
;; Private Functions
;;;;;;;;;;;;;;;;

(define dummy-pair (cons #f #f))
(define (node-output-set! node new-output)
  (let ([old-output (flow-graph-node-output node)])
    (when (not ((flow-graph-node-eqv? node) old-output new-output))
      #;(flow-graph-node-outputs-set!
          node (cons old-output (flow-graph-node-outputs node)))
      (flow-graph-node-output-set! node new-output)
      (let loop ([list (flow-graph-node-listeners node)]
                 [prev dummy-pair]) ;; just to get the loop started
        (if (null? list)
            (values)
```

```
            (let ([x (car list)]
                  [rest (cdr list)])
              (if (not (pair? x))
                  (add-to-pending! x)
                  (let ([car-x (car x)])
                    (if car-x
                        (add-to-pending! car-x)
                        (set-cdr! prev rest))))
              (loop rest list))))
      (set-cdr! dummy-pair #f))))

(define (add-to-pending! node)
  (assert (not (eq? (flow-graph-node-action node) flow-graph-mutable-node-action)))
  (flow-graph-node-pending-set! node #t)
  (flow-graph-pending-add! (flow-graph-node-graph node) node))

(define (add-to-listeners! node listener)
  (flow-graph-node-listeners-set!
   node (cons listener (flow-graph-node-listeners node))))

(define (add-to-join-listeners! node listener)
  (let ([key (list listener)])
    (add-to-listeners! node key)
    key))

(define (remove-from-join-listeners! key)
  (set-car! key #f))

(record-writer (record-type-descriptor flow-graph-node)
  (lambda (r p wr)
    (display "#[node␣" p)
    (display (flow-graph-node-info r) p)
    (display "␣" p)
    (display (flow-graph-node-output r) p)
    (display "]" p)))

)
```

Listing B.17: ast-flow-graphs.ss

```
(library (ast-flow-graphs)
  (export
   with-AST-flow-graph
   AST-flow-graph ;; TODO: restrict this interface
   make-AST-node
   AST-flow-graph-print
#|
   ;; DEBUG ONLY
   with-local-Formal-in-type
   with-local-Formal-out-bitype
   Formal-in-type-set!
   Formal-out-bitype-set!
|#
   )
  (import (except (rnrs) case cond if) (safe-forms) (srfi-39) (flow-graphs)
          (ast) (idents) (types) (bitypes))
```

```
;; make-AST-* versions expect nodes as argument
;; AST->* versions expect ASTs as argument

#|
;; DEBUG ONLY (nodes)
(define-hashtable-parameter with-local-Formal-in-type
  Formal-in-type Formal-in-type-set!)
(define-hashtable-parameter with-local-Formal-out-bitype
  Formal-out-bitype Formal-out-bitype-set!)
|#

(define AST-flow-graph-parameter (make-parameter #f))
(define (AST-flow-graph) (AST-flow-graph-parameter))

(define-syntax with-AST-flow-graph
  (syntax-rules ()
    [(_ graph expr0 expr* ...)
     (parameterize ([AST-flow-graph-parameter graph]) expr0 expr* ...)]))

(define-syntax make-AST-node
  (syntax-rules ()
    [(_ ast make arguments ...)
     (let ([ast-var ast])
       (assert (or (AST? ast-var) (case-lambda-clause? ast-var)))
       (let ([node (make (AST-flow-graph) arguments ...)])
         (AST-contained-nodes-add! ast-var node)
         node))]))

(define (AST-contained-nodes-add! ast node)
  (cond
   [(AST? ast)
    (AST-contained-nodes-set! ast (cons node (AST-contained-nodes ast)))]
   [(case-lambda-clause? ast)
    (case-lambda-clause-contained-nodes-set!
     ast (cons node (case-lambda-clause-contained-nodes ast)))]))

(define (AST-flow-graph-print ast)
  (define (obj->string obj)
    (call-with-string-output-port (lambda (port) (write obj port))))

  (define (ident->string ident)
    (string-append "[" (obj->string (ident-name ident))
                   "␣" (obj->string (ident-number ident))
                   "␣" (obj->string (ident-assigned ident)) "]"))

  ;; contain : string * list of node * list of ast
  (define (printer contain ast)
    (cond
     [(case-lambda-clause? ast)
      (contain
       (string-append
        "(Clause␣"
        (let ([formals (vector->list (case-lambda-clause-formals ast))])
          (string-append
           "("
           (apply string-append (map ident->string formals))
;;             ";"
;;             (apply string-append
;;                    (map obj->string
```

```
;;                          (map flow-graph-node-output
;;                               (map Formal-in-type formals (map (lambda (x) #f)
;;                                                                formals)))))
;;              ";"
;;              (apply string-append
;;                     (map obj->string
;;                          (map flow-graph-node-output
;;                               (map Formal-out-bitype formals (map (lambda (x) #f)
;;                                                                   formals)))))
           ")"))
        (let ([rest (case-lambda-clause-rest ast)])
          (if rest (ident->string rest) "#f"))
        ")")
       (case-lambda-clause-contained-nodes ast)
       (list (case-lambda-clause-body ast)))]
     [else
      (contain
       (string-append
        (AST-case ast
          [(Ref var) (string-append "(Ref␣" (ident->string var) ")")]
          [(Quote obj) (string-append "(Quote␣" (obj->string obj) ")")]
          [(Primref safe name) (string-append "(Primref␣" (if safe "#t" "#f")
                                              "␣" (symbol->string name) ")")]
          [(Set var body) (string-append "(Set␣" (ident->string var) ")")]
          [(App fun args) "(App)"]
          [(If test conseq altern) "(If)"]
          [(CaseLambda table clauses) (string-append "(CaseLambda␣"
                                                     (obj->string table) ")")]
          [(Seq e1 e2) "(Seq)"]
          [(Letrec bindings body)
           (string-append
            "(Letrec␣"
            (apply string-append (vector->list
                                  (vector-map
                                   ident->string (vector-map
                                                  letrec-binding-lhs bindings))))
            ")")])
        ":"
        (obj->string (flow-graph-node-output (AST-output-node ast))))
       (AST-contained-nodes ast)
       (AST-case ast
         [(Ref var) '()]
         [(Quote obj) '()]
         [(Primref safe name) '()]
         [(Set var body) (list body)]
         [(App fun args) (cons fun (vector->list args))]
         [(If test conseq altern) (list test conseq altern)]
         [(CaseLambda table clauses) (vector->list clauses)]
         [(Seq e1 e2) (list e1 e2)]
         [(Letrec bindings body)
          (append (map letrec-binding-rhs (vector->list bindings)) (list body))]))])
             )

  (define (labeler node)
    (string-append
     (flow-graph-node-info node) ":" (obj->string (flow-graph-node-output node))))
  (flow-graph-print printer labeler ast))

)
```

B.3.3. Types

Listing B.18: types.ss

```
(library (types)
  (export
   select-CFA
   type? ;; obj -> bool
   type=? ;; type * type -> bool
   type<=? ;; type * type -> bool (true if the type is = or at a lower point on the
       lattice)
  proc-type-bits=? ;; list-of type * list-of type -> bool (true if the first and
      second set of procs contain the same information)

  ;; non-proc type bits (fixnum)
  non-proc-type-bits
  other-type-bits
  false-type-bits
  true-type-bits
  pair-or-null-type-bits
  null-type-bits
  pair-type-bits
  symbol-type-bits
  boolean-type-bits
  number-type-bits
  fxnum-type-bits
  flonum-type-bits
  other-number-type-bits
  exact-integer-type-bits
  fxzero-type-bits
  bignum-type-bits
  length-type-bits
  vector-type-bits
  string-type-bits
  char-type-bits

  ;; proc-type-bits
  type-procs ;; type -> proc-type-bits
  make-proc-type
  make-proc-record
  proc-type-bits-case
  bot-proc-type-bits ;; void -> proc-type-bits
  top-proc-type-bits ;; void -> proc-type-bits
  one-or-more-proc-type-bits ;; proc-type -> proc-type-bits

  ;; proc-record
  proc-record->type ;; proc-record -> type
  proc-record-input-called-proc ;; proc-record -> ???
  proc-record-input-argument-proc ;; proc-record -> ???
  proc-record-output-return-proc ;; proc-record -> ???
  proc-record-output-argument-proc ;; proc-record -> ???

  ;; top/bot related types
  top-type ;; type
  bot-type ;; type
  bot-type? ;; type -> bool
  non-bot-type? ;; type -> bool
```

```
  ;; type predicates and constants
  proc-type ;; type
  true-type true-type? ;; type, type->bool
  false-type false-type? ;; type, type->bool
  pair-type? null-type? ;; type, type->bool

  ;; type operators (type * type -> type)
  type-union
  type-intersect

  maybe-non-proc-type->type
  not-type
  type-of
  mark-escaped-type!
  type-filter
  )
 (import (except (rnrs) case cond if) (safe-forms) (records-io) (datatype)
         (only (chezscheme) void bignum?))

;; Choose to use 0CFA or Sub-0CFA at compile-time and avoid run-time dispatch
(define-syntax select-CFA
  (syntax-rules (zero-CFA sub-zero-CFA)
    [(_ [(zero-CFA) zcfa zcfa* ...] [(sub-zero-CFA) sub-zcfa sub-zcfa* ...])
     (begin sub-zcfa sub-zcfa* ...)]   ;; change to sub-zcfa to choose sub-zero CFA
    [(_ [(sub-zero-CFA) sub-zcfa sub-zcfa* ...] [(zero-CFA) zcfa zcfa* ...])
     (begin sub-zcfa sub-zcfa* ...)])) ;; change to sub-zcfa to choose sub-zero CFA

;; Full types
(define-record-type type (fields non-proc procs)
  (nongenerative)
  (protocol
   (lambda (make)
     (lambda (non-proc procs)
       (assert (fixnum? non-proc))
       (assert (proc-type-bits? procs))
       (make non-proc procs)))))

(define (type=? x y)
  (and (= (type-non-proc x) (type-non-proc y))
       (proc-type-bits=? (type-procs x) (type-procs y))))

(define (type<=? x y)
  (or (type=? x y) (type=? x (type-intersect x y))))

;; Non-proc types
(define-syntax flatten-mask-bits ;; (a b (c d e)) -> (bitwise-ior a b c d e)
  (syntax-rules ()
    [(_ (arg sub-args ...) args ...)
     (bitwise-ior arg (flatten-mask-bits args ...))]
    [(_ arg args ...)
     (bitwise-ior arg (flatten-mask-bits args ...))]
    [(_) 0]))

(define-syntax define-mask-bits
  (syntax-rules ()
    [(_ offset (arg sub-args ...) args ...)
     (begin
       (define-mask-bits offset sub-args ... args ...)
```

```
      (define arg (flatten-mask-bits sub-args ...)))]
    [(_ offset arg) (define arg (bitwise-arithmetic-shift 1 offset))]
    [(_ offset arg args ...)
     (begin
       (define arg (bitwise-arithmetic-shift 1 offset))
       (define-mask-bits (+ 1 offset) args ...))]))

(define-mask-bits 0
  (non-proc-type-bits

   ;; Singleton types
   (false-non-proc-type-bits
    false-type-bits)

   (true-non-proc-type-bits
    other-type-bits ;; handles types not listed here (e.g. unspecified type)
    true-type-bits

    ;; Container types
    (pair-or-null-type-bits
     null-type-bits ;; not exactly a container type, but pair-or-null-type is.
     pair-type-bits)
    vector-type-bits

    ;; Atomic types
    string-type-bits
    char-type-bits
    symbol-type-bits ;; TODO: property lists
    (number-type-bits
     (other-number-type-bits
      flonum-type-bits)
     (exact-integer-type-bits
      (bignum-type-bits
       neg-bignum-type-bits
       pos-bignum-type-bits)
      (fxnum-type-bits
       min-fxnum-type-bits
       neg-index-type-bits
       neg-one-type-bits
       (length-type-bits
        fxzero-type-bits
        one-type-bits
        index-type-bits
        max-fxnum-type-bits)))))))

(define boolean-type-bits (bitwise-ior true-type-bits false-type-bits))

;; Proc types

;; For type that hasn't yet been discovered to be proc.
(define-syntax bot-proc-type-bits
  (select-CFA
   [(zero-CFA) (syntax-rules () [(_) '()])]
   [(sub-zero-CFA) (syntax-rules () [(_) #f])]))

;; For procedure that has been discovered to be the union of two
;; different procs.
(define-syntax top-proc-type-bits
  (select-CFA
```

```
    [(zero-CFA) (syntax-rules () [(_) (list #t)])]
    [(sub-zero-CFA) (syntax-rules () [(_) #t])]))

;; Other proc types point directly to a lambda box
(define-syntax one-or-more-proc-type-bits
  (select-CFA
   [(zero-CFA) (syntax-rules () [(_ x x* ...) (list x x* ...)])]
   [(sub-zero-CFA) (syntax-rules () [(_ x) x])]))

(define (proc-type-bits? x)
  (select-CFA
   [(zero-CFA) (for-all (lambda (x) (or (proc-record? x) (eq? #t x))) x)]
   [(sub-zero-CFA) (or (eq? x #t) (eq? x #f) (proc-record? x))]))

(define-syntax proc-type-bits-case
  (syntax-rules (bot-proc-type-bits top-proc-type-bits one-or-more-proc-type-bits)
    [(_ proc-type-bits
        [(bot-proc-type-bits) bot ...]
        [(top-proc-type-bits) top ...]
        [(one-or-more-proc-type-bits procs) one-or-more ...])
     (select-CFA
      [(zero-CFA)
       (let ([procs proc-type-bits])
         (cond
          [(eq? procs '()) bot ...]
          [(memq #t procs) top ...]
          [else one-or-more ...]))]
      [(sub-zero-CFA)
       (let ([procs proc-type-bits])
         (cond
          [(eq? procs #f) bot ...]
          [(eq? procs #t) top ...]
          [else one-or-more ...]))])]))

(define (proc-type-bits=? x y)
  (select-CFA
   [(zero-CFA)
    ;; ugly -- doing the quadratic check to see if two sets of procs are the same.
    (let f ([x* x] [y* y] [ry* '()] [looped? #f])
      (cond
       [(and (null? x*) (null? y*)) (null? ry*)) #t]
       [(or (null? x*) (and (null? y*) (null? ry*)) (and (null? y*) looped?)) #f]
       [(null? y*) (f x* ry* '() #t)]
       [(eq? (car x*) (car y*)) (f (cdr x*) (cdr y*) ry* #f)]
       [else (f x* (cdr y*) (cons (car y*) ry*) looped?)]))]
   [(sub-zero-CFA) (eq? x y)]))

(define-record-type proc-record
  (nongenerative)
  (fields
   escape-proc ;; void -> void
   input-called-proc ;; arg-count * bool -> void
   input-argument-proc ;; arg-count * position * type -> void
   output-return-proc ;; arg-count -> flow-graph-node[type]
   output-argument-proc ;; arg-count * position -> flow-graph-node[bitype]
   ))

(define (make-proc-type escape
                        input-called input-argument
```

```
                        output-return output-argument)
  (make-type 0 (one-or-more-proc-type-bits
                (make-proc-record
                 escape input-called input-argument
                 output-return output-argument))))

(define (proc-record->type x) (make-type 0 x))

;; Type constants and predicates
(define top-type (make-type non-proc-type-bits (top-proc-type-bits)))
(define bot-type (make-type 0 (bot-proc-type-bits)))
(define (bot-type? x) (assert (type? x)) (type=? bot-type x))
(define (non-bot-type? x) (assert (type? x)) (not (bot-type? x)))

(define proc-type (make-type 0 (top-proc-type-bits)))
(define true-type (make-type true-non-proc-type-bits (top-proc-type-bits)))
(define (true-type? x) (non-bot-type? (type-intersect x true-type)))
(define false-type (make-type false-non-proc-type-bits (bot-proc-type-bits)))
(define (false-type? x) (non-bot-type? (type-intersect x false-type)))

(define pair-type (make-type pair-type-bits (bot-proc-type-bits)))
(define (pair-type? x) (non-bot-type? (type-intersect x pair-type)))
(define null-type (make-type null-type-bits (bot-proc-type-bits)))
(define (null-type? x) (non-bot-type? (type-intersect x null-type)))

(define (type-intersect x y)
  (make-type
   (bitwise-and (type-non-proc x) (type-non-proc y))
   (select-CFA
    [(zero-CFA) (proc-type-bits-intersect (type-procs x) (type-procs y))]
    [(sub-zero-CFA)
     (proc-type-bits-case (type-procs x)
       [(bot-proc-type-bits) (bot-proc-type-bits)]
       [(top-proc-type-bits) (type-procs y)]
       [(one-or-more-proc-type-bits proc-x)
        (proc-type-bits-case (type-procs y)
          [(bot-proc-type-bits) (bot-proc-type-bits)]
          [(top-proc-type-bits) (type-procs x)]
          [(one-or-more-proc-type-bits proc-y)
           (if (eq? proc-x proc-y) (type-procs x) (bot-proc-type-bits))])])])))

(define (proc-type-bits-intersect x* y*)
  (cond
   [(memq #t x*) y*]
   [(memq #t y*) x*]
   [else (let f ([x* x*] [final* '()])
           (if (null? x*)
               final*
               (let ([x (car x*)])
                 (if (memq x y*)
                     (f (cdr x*) (cons x final*))
                     (f (cdr x*) final*)))))]))

(define (proc-type-bits-union x* y*)
  (let f ([x* x*] [xy* y*])
    (if (null? x*)
        xy*
        (let ([x (car x*)])
          (if (memq x y*)
```

```
              (f (cdr x*) xy*)
              (f (cdr x*) (cons x xy*)))))))

;; NOTE: may side-effect escape value of procs
(define (type-union x y)
  (select-CFA
   [(zero-CFA)
    (define (mark-escaped-lambda! proc*)
      (map (lambda (proc) (unless (eq? proc #t) ((proc-record-escape-proc proc))))
           proc*))
    (let ([proc-x* (type-procs x)] [proc-y* (type-procs y)])
      (when (memq #t proc-x*) (mark-escaped-lambda! proc-y*))
      (when (memq #t proc-y*) (mark-escaped-lambda! proc-x*)))
    (make-type
     (bitwise-ior (type-non-proc x) (type-non-proc y))
     (proc-type-bits-union (type-procs x) (type-procs y)))]
   [(sub-zero-CFA)
    (define (mark-escaped-lambda! proc) ((proc-record-escape-proc proc)))
    (make-type
     (bitwise-ior (type-non-proc x) (type-non-proc y))
     (proc-type-bits-case (type-procs x)
       [(bot-proc-type-bits) (type-procs y)]
       [(top-proc-type-bits)
        (proc-type-bits-case (type-procs y)
          [(bot-proc-type-bits) (values)]
          [(top-proc-type-bits) (values)]
          [(one-or-more-proc-type-bits proc-y) (mark-escaped-lambda! proc-y)])
        (top-proc-type-bits)]
       [(one-or-more-proc-type-bits proc-x)
        (proc-type-bits-case (type-procs y)
          [(bot-proc-type-bits) (type-procs x)]
          [(top-proc-type-bits) (mark-escaped-lambda! proc-x) (top-proc-type-bits)]
          [(one-or-more-proc-type-bits proc-y)
           (if (eq? proc-x proc-y)
               (type-procs x)
               (begin
                 (mark-escaped-lambda! proc-x) ;; rather then escaping keep both
                 (mark-escaped-lambda! proc-y)
                 (top-proc-type-bits)))])]))]))

(define (mark-escaped-type! type)
  (select-CFA
   [(zero-CFA)
    (for-each
     (lambda (proc) (when (proc-record? proc) ((proc-record-escape-proc proc))))
     (type-procs type))]
   [(sub-zero-CFA) (type-union type top-type)])
  (values))

(define (type-filter type true false)
  (type-union
   (if true (type-intersect type true-type) bot-type)
   (if false (type-intersect type false-type) bot-type)))

;; helpers for prims
;; TODO: rename as type-complement
(define (not-type non-proc-type-bits)
  (make-type (bitwise-not non-proc-type-bits) (top-proc-type-bits)))
;; TODO: make work for proc-type
```

```
(define (maybe-non-proc-type->type type)
  (if (type? type) type (make-type type (bot-proc-type-bits))))

(define (type-of x)
  (make-type
   (cond
    [(eq? x (void)) other-type-bits]
    [(eq? x #f) false-type-bits]
    [(eq? x #t) true-type-bits]
    [(eq? x '()) null-type-bits]
    [(pair? x) pair-type-bits]
    [(vector? x) vector-type-bits]
    [(symbol? x) symbol-type-bits]
    [(eq? x 0) fxzero-type-bits]
    [(eq? x 1) one-type-bits]
    [(eq? x -1) neg-one-type-bits]
    [(and (not (bignum? x)) (fixnum? x) (bignum? (+ x 1))) max-fxnum-type-bits]
    [(and (not (bignum? x)) (fixnum? x) (bignum? (- x 1))) min-fxnum-type-bits]
    [(and (fixnum? x) (positive? x)) index-type-bits]
    [(and (fixnum? x) (negative? x)) neg-index-type-bits]
    [(and (bignum? x) (positive? x)) pos-bignum-type-bits]
    [(and (bignum? x) (negative? x)) neg-bignum-type-bits]
    [(string? x) string-type-bits]
    [(procedure? x) 0]
    [(flonum? x) flonum-type-bits]
    [(char? x) char-type-bits]
    [else non-proc-type-bits])
   (if (procedure? x) (top-proc-type-bits) (bot-proc-type-bits))))

(record-writer (record-type-descriptor type)
  (lambda (r p wr)
    (display "#<0x" p)
    (display (number->string (type-non-proc r) 16) p)
    (display ", " p)
    (wr (proc-type-bits-case (type-procs r)
          [(bot-proc-type-bits) 'bot]
          [(top-proc-type-bits) 'top]
          [(one-or-more-proc-type-bits proc) 'one-or-more-proc]) p)
    (display ">" p)))

(record-writer (record-type-descriptor proc-record)
  (lambda (r p wr) (display "#[proc-type]" p)))

)
```

Listing B.19: bitypes.ss

```
(library (bitypes)
  (export
   make-bitype ;; type * type -> bitype
   bitype? ;; obj -> bool
   bitype=? ;; bitype * bitype -> bool
   bitype-true ;; bitype -> type
   bitype-false ;; bitype -> type

   bitype->type ;; bitype into join of true and false
   type->bitype ;; type into true and false
```

```
    type->bitype-true ;; type into true only
    type->split-bitype ;; true values into true and false values into false

    bitype->void-bitype ;; true and false go to both true and false
    bitype-true->void-bitype ;; only true goes to true and/or false
    bitype-false->void-bitype ;; only false goes to true and/or false

    bitype-union ;; bitype * bitype -> bitype
    bitype-intersect ;; bitype * bitype -> bitype
    )
  (import (rnrs) (types) (records-io))

(define-record-type bitype
  (nongenerative)
  (fields true false)
  (protocol
   (lambda (make-bitype)
     (lambda (true false)
       (assert (type? true))
       (assert (type? false))
       (make-bitype true false)))))

(define (bitype=? x y)
  (and (type=? (bitype-true x) (bitype-true y))
       (type=? (bitype-false x) (bitype-false y))))

(define (bitype->type bitype)
  (type-union (bitype-true bitype) (bitype-false bitype)))

(define (type->bitype type) (make-bitype type type))
(define (type->bitype-true type) (make-bitype type bot-type))

(define (type->split-bitype type)
  (make-bitype (type-intersect type true-type)
               (type-intersect type false-type)))

(define (bitype->void-bitype bitype) (type->bitype (bitype->type bitype)))

(define (bitype-true->void-bitype bitype) (type->bitype (bitype-true bitype)))
(define (bitype-false->void-bitype bitype) (type->bitype (bitype-false bitype)))

(define (bitype-union t1 t2)
  (make-bitype (type-union (bitype-true t1) (bitype-true t2))
               (type-union (bitype-false t1) (bitype-false t2))))

(define (bitype-intersect t1 t2)
  (make-bitype (type-intersect (bitype-true t1) (bitype-true t2))
               (type-intersect (bitype-false t1) (bitype-false t2))))

(record-writer (record-type-descriptor bitype)
  (lambda (r p wr)
    (display "#<" p)
    (wr (bitype-true r) p)
    (display "/" p)
    (wr (bitype-false r) p)
    (display ">" p)))

)
```

Listing B.20: env-flow-functions.ss

```
(library (env-flow-functions)
  (export
   cond-env-flow-function ;; macro:
                          ;;   (_ ([test conseq ...]) ([test conseq] ...))
                          ;;   -> env-flow-function
   env-flow-function=? ;; env-flow-function * env-flow-function -> bool
   env-flow-function-zero ;; env-flow-function
   env-flow-function-identity ;; env-flow-function
   env-flow-function-compose ;; env-flow-function *
                             ;; env-flow-function ->
                             ;; env-flow-function (associative)
   env-flow-function-apply ;; env-flow-function * bitype * type -> bitype
   true false bypass ;; aux keywords
   )
  (import (rnrs) (types) (bitypes) (records-io))

(define true 0) ;; aux keyword, value doesn't matter
(define false 0) ;; aux keyword, value doesn't matter
(define bypass 0) ;; aux keyword, value doesn't matter

;; Constants for the bit level encoding
(define true-base 4)
(define false-base 0)
(define true-true-bit (+ true-base 2))
(define true-false-bit (+ true-base 1))
(define true-bypass-bit (+ true-base 0))
(define false-true-bit (+ false-base 2))
(define false-false-bit (+ false-base 1))
(define false-bypass-bit (+ false-base 0))
(define true-mask (fxior (fxarithmetic-shift 1 true-true-bit)
                         (fxarithmetic-shift 1 true-false-bit)
                         (fxarithmetic-shift 1 true-bypass-bit)))
(define false-mask (fxior (fxarithmetic-shift 1 false-true-bit)
                          (fxarithmetic-shift 1 false-false-bit)
                          (fxarithmetic-shift 1 false-bypass-bit)))
(define bypass-mask (fxior (fxarithmetic-shift 1 true-bypass-bit)
                           (fxarithmetic-shift 1 false-bypass-bit)))

(define-syntax flow-mask
  (syntax-rules (true false bypass)
    [(_ true true) (fxarithmetic-shift 1 true-true-bit)]
    [(_ true false) (fxarithmetic-shift 1 true-false-bit)]
    [(_ true bypass) (fxarithmetic-shift 1 true-bypass-bit)]
    [(_ false true) (fxarithmetic-shift 1 false-true-bit)]
    [(_ false false) (fxarithmetic-shift 1 false-false-bit)]
    [(_ false bypass) (fxarithmetic-shift 1 false-bypass-bit)]))

(define-syntax flow-masks
  (syntax-rules ()
    [(_ _) 0]
    [(_ out in) (flow-mask out in)]
    [(_ out in ins ...) (fxior (flow-mask out in) (flow-masks out ins ...))]))

(define-syntax $make-env-flow-source
  (syntax-rules ()
    [(_ _) 0]
```

```
    [(_ out [test conseq]) (if test (flow-masks out conseq) 0)]
    [(_ out [test conseq] . rest)
     (fxior (if test (flow-masks out conseq) 0)
            ($make-env-flow-source out . rest))]))

(define-syntax cond-env-flow-function
  (syntax-rules ()
    [(_ ([true-test true-conseq] ...) ([false-test false-conseq] ...))
     (fxior
      ($make-env-flow-source true [true-test true-conseq] ...)
      ($make-env-flow-source false [false-test false-conseq] ...))]))

(define env-flow-function-zero 0)

(define env-flow-function-identity
  (fxior (flow-mask true true) (flow-mask false false)))

;; env-flow-function * env-flow-function -> bool
(define (env-flow-function=? x y) (fx=? x y))

;; env-flow-function * env-flow-function -> env-flow-function
;; f * g -> (f.g)
(define (env-flow-function-compose outer inner)
  (fxior
   (fxior ;; WORKAROUND: Chez does strange allocation if we don't do extra fxior
    (fxand
     inner
     (if (fxbit-set? outer true-true-bit)
         (if (fxbit-set? outer false-false-bit)
             (fxior true-mask false-mask) true-mask)
         (if (fxbit-set? outer false-false-bit)
             false-mask 0)))
    (if (fxbit-set? outer true-false-bit)
        (fxand true-mask (fxarithmetic-shift inner (- true-base false-base)))
        0)
    (if (fxbit-set? outer false-true-bit)
        (fxand false-mask (fxarithmetic-shift inner (- false-base true-base)))
        0))
   (fxand bypass-mask outer)))

;; env-flow-function * bitype * type -> bitype
(define (env-flow-function-apply function bitype bypass)
  (define (env-flow-source-apply source)
    (define t1 (if (fxbit-set? source true-true-bit)
                   (bitype-true bitype)
                   bot-type))
    (define t2 (if (fxbit-set? source true-false-bit)
                   (type-union t1 (bitype-false bitype))
                   t1))
    (define t3 (if (fxbit-set? source true-bypass-bit)
                   (type-union t2 bypass)
                   t2))
    t3)
  (assert (bitype? bitype))
  (assert (type? bypass))

  (make-bitype
   (env-flow-source-apply function)
   (env-flow-source-apply (fxarithmetic-shift function (- true-base false-base)))))
```

```
)
```

Listing B.21: prims.ss

```
(library (prims)
  (export prim-name->type)
  (import (except (rnrs) case cond if) (safe-forms) (idents) (types) (bitypes)
          (syntax-helpers) (ast-flow-graphs) (flow-graphs))

(define (make-AST-const-node ast name const)
  (make-AST-node ast make-flow-graph-mutable-node name eq? const))

(define-syntax make-prim-core
  (lambda (stx)
    (syntax-case stx ()
      [(_ name [(return-true return-false)
                (arg-escapes arg-true arg-false) ...
                (has-rest rest-escapes rest-true rest-false)] ...)
       (with-syntax ([(return ...)
                      (generate-temporaries #'(return-true ...))]
                     [((arg ...) ...)
                      (syntax-map generate-temporaries #'((arg-true ...) ...))]
                     [(rest ...)
                      (generate-temporaries #'(rest-true ...))]
                     [((formal-position ...) ...)
                      (syntax-map syntax-indexes #'((arg-true ...) ...))]
                     [(clause-arg-count ...)
                      (syntax-map syntax-length #'((arg-true ...) ...))]
                     )
         #'(cons
            'name
            (let ([name
                   (lambda (ast)
                     (define bot
                       (make-AST-const-node ast "no-return" bot-type))
                     (define bibot
                       (make-AST-const-node ast "no-return" (type->bitype bot-type))
                           )
                     (define return
                       (make-AST-const-node ast "return"
                         (type-union (maybe-non-proc-type->type return-true)
                                     (maybe-non-proc-type->type return-false)))) ...
                     (define arg
                       (make-AST-const-node
                        ast "arg"
                        (make-bitype (maybe-non-proc-type->type arg-true)
                                     (maybe-non-proc-type->type arg-false)))) ...
                                         ...
                     (define rest
                       (make-AST-const-node
                        ast "rest" (make-bitype
                                    (maybe-non-proc-type->type rest-true)
                                    (maybe-non-proc-type->type rest-false)))) ...
                     (make-proc-type
                      (lambda () '()) ;; escape
                      (lambda (arg-count called) '()) ;; input-called
```

```
                      (lambda (arg-count position type) ;; input-argument
                        (when (or (and
                                   ((if has-rest >= =) arg-count clause-arg-count)
                                   (or (and arg-escapes (= position formal-position)
                                            )
                                       ...
                                       (and has-rest rest-escapes
                                            (>= position clause-arg-count))))
                                  ...)
                          (mark-escaped-type! type)
                          #f)) ;; TODO: void instead of #f
                      (lambda (arg-count) ;; output-return
                        (cond
                         [((if has-rest >= =) arg-count clause-arg-count)
                          return]
                         ...
                         [else bot]))
                      (lambda (arg-count position) ;; output-argument
                        (cond
                         [((if has-rest >= =) arg-count clause-arg-count)
                          (cond
                           [(= position formal-position) arg]
                           ...
                           [(and has-rest (>= position clause-arg-count)) rest]
                           [else bibot])]
                         ...
                         [else bibot])))])
              name)))])))

(define-syntax make-prim-rest
  (lambda (stx)
    (define (add-rest stx)
      (syntax-case stx ()
        [((return-true return-false) args ... (escapes true false) dots)
         (syntax-ellipsis? #'dots)
         #'((return-true return-false) args ... (#t escapes true false))]
        [((return-true return-false) args ... (dots true false))
         (syntax-ellipsis? #'dots)
         ;; Fix problem with make-top-prim
         (add-rest #'((return-true return-false) args ... dots))]
        [((return-true return-false) args ...)
         #'((return-true return-false) args ... (#f #f bot-type bot-type))]))
    (syntax-case stx ()
      [(_ name clauses ...)
       #`(make-prim-core name #,@(syntax-map add-rest #'(clauses ...)))])))

(define-syntax make-prim
  (syntax-rules ()
    [(_ (names ...) clauses ...) (list (make-prim-rest names clauses ...) ...)]))

(define-syntax make-top-prim
  (syntax-rules ()
    [(_ (names ...) [(true-type false-type) arg-escapes ...] ...)
     (make-prim (names ...)
                [(true-type false-type) (arg-escapes top-type top-type) ...]
                ...)]))

(define-syntax make-pred-prim
  (syntax-rules ()
```

```
    [(_ (names ...) type)
     (make-prim (names ...)
                [(true-type-bits false-type-bits) (#f type (not-type type))])]))

;; TODO: expand arithmetic (do as wrapper around make-prim?)
;;   (may be unneeded in new design b/c proc-type is used instead of case-lambda)
;;   (same logic for in argument position?)
;; NOTE: for non-smart prims argument output types are
;;   the return types assuming prim has escaped
;; TODO: case for bare vs in application

;; TODO: as a hashtable
(define prim-table
  (append

   (make-prim (foreign-procedure) [(proc-type bot-type)
                                   (#f string-type-bits top-type)])

   (make-prim (foreign-callable) [(other-type-bits ;; code object
                                   bot-type)
                                  (#t proc-type non-proc-type-bits)])

   (make-prim (error)
     [(bot-type bot-type)
      (#f symbol-type-bits symbol-type-bits)
      (#f string-type-bits string-type-bits)
      (#f top-type top-type) ...])

   ;; TODO: escape should return the type of its argument
   (make-top-prim (escape) [(top-type top-type) #t])

   (make-prim (for-all)
     [(true-type false-type-bits)
      (#t proc-type proc-type)
      (#f pair-or-null-type-bits pair-or-null-type-bits)
      (#f pair-or-null-type-bits pair-or-null-type-bits) ...])

   ;; pair related primitives
   (make-pred-prim (pair?) pair-type-bits)

   ;; on false return could still be a pair b/c of improper lists
   (make-prim (list?)
     [(true-type-bits false-type-bits)
      (#f pair-or-null-type-bits (not-type null-type-bits))])

   (make-top-prim (cons) [(pair-type-bits bot-type) #t #t])

   (make-top-prim (list) [(pair-type-bits bot-type) #t ...])

   (make-prim (car cdr) [(true-type false-type)
                         (#f pair-type-bits pair-type-bits)])

   (make-prim (for-each)
     [(top-type top-type)
      (#t proc-type proc-type)
      (#f pair-or-null-type-bits pair-or-null-type-bits)
      (#f pair-or-null-type-bits pair-or-null-type-bits) ...])

   (make-prim (map)
```

```
    [(pair-or-null-type-bits bot-type)
     (#t proc-type proc-type)
     (#f pair-or-null-type-bits pair-or-null-type-bits)
     (#f pair-or-null-type-bits pair-or-null-type-bits) ...])

(make-prim (reverse)
  [(pair-or-null-type-bits bot-type)
   (#f pair-or-null-type-bits pair-or-null-type-bits)])

(make-prim (list->vector)
  [(vector-type-bits bot-type)
   (#f pair-or-null-type-bits pair-or-null-type-bits)])

(make-prim (r6rs:number->string number->string)
  [(string-type-bits bot-type)
   (#f number-type-bits number-type-bits)]
  [(string-type-bits bot-type)
   (#f number-type-bits number-type-bits)
   (#f fxnum-type-bits fxnum-type-bits)]
  [(string-type-bits bot-type)
   (#f number-type-bits number-type-bits)
   (#f fxnum-type-bits fxnum-type-bits)
   (#f fxnum-type-bits fxnum-type-bits)])

(make-prim (r6rs:string->number string->number)
  [(top-type bot-type)                    ; maybe-number, no way to express it
   (#f string-type-bits string-type-bits)]
  [(top-type bot-type)
   (#f string-type-bits string-type-bits)
   (#f fxnum-type-bits fxnum-type-bits)])

(make-prim (char?)
  [(true-type-bits false-type-bits)
   (#f char-type-bits (not-type char-type-bits))])

(make-prim (put-char)
  [(other-type-bits bot-type)
   (#f other-type-bits other-type-bits)
   (#f char-type-bits char-type-bits)])

(make-prim (write-char)
  [(other-type-bits bot-type)
   (#f char-type-bits char-type-bits)]
  [(other-type-bits bot-type)
   (#f char-type-bits char-type-bits)
   (#f other-type-bits other-type-bits)])

(make-prim (char->integer)
  [(fxnum-type-bits bot-type)
   (#f char-type-bits char-type-bits)])

(make-prim (char-alphabetic? char-numeric? char-whitespace? char-upper-case?
            char-lower-case? char-title-case?)
  [(true-type-bits false-type-bits)
   (#f char-type-bits char-type-bits)])

(make-prim (char-general-category)
  [(symbol-type-bits bot-type)
   (#f char-type-bits char-type-bits)])
```

```
(make-prim (integer->char)
  [(char-type-bits bot-type)
   (#f fxnum-type-bits fxnum-type-bits)])

(make-prim (char=? char<=? char<? char>? char>=?)
  [(true-type-bits false-type-bits)
   (#f char-type-bits char-type-bits)
   (#f char-type-bits char-type-bits) ...])

(make-prim (char-upcase char-downcase char-titlecase char-foldcase)
  [(char-type-bits bot-type)
   (#f char-type-bits char-type-bits)])

(make-prim (string?)
  [(true-type-bits false-type-bits)
   (#f string-type-bits (not-type string-type-bits))])

(make-prim (open-string-input-port)
  [(other-type-bits bot-type)
   (#f string-type-bits string-type-bits)])

(make-prim (list->string)
  [(string-type-bits bot-type)
   (#f pair-or-null-type-bits pair-or-null-type-bits)])

(make-prim (string->list)
  [(pair-or-null-type-bits bot-type)
   (#f string-type-bits string-type-bits)])

(make-prim (string-append)
  [(string-type-bits bot-type)
   (#f string-type-bits string-type-bits) ...])

(make-prim (string-length)
  [(fxnum-type-bits bot-type)
   (#f string-type-bits string-type-bits)])

(make-prim (string-ref)
  [(char-type-bits bot-type)
   (#f string-type-bits string-type-bits)
   (#f fxnum-type-bits fxnum-type-bits)])

(make-prim (string-set!)
  [(true-type bot-type)
   (#f string-type-bits string-type-bits)
   (#f fxnum-type-bits fxnum-type-bits)
   (#f char-type-bits char-type-bits)])

(make-prim (string)
  [(string-type-bits bot-type)
   (#f char-type-bits char-type-bits) ...])

(make-prim (r6rs:char=? r6rs:char<=? r6rs:char<? r6rs:char>? r6rs:char>=?)
  [(true-type-bits false-type-bits)
   (#f char-type-bits char-type-bits)
   (#f char-type-bits char-type-bits)
   (#f char-type-bits char-type-bits) ...])
```

```
(make-prim (r6rs:string=? r6rs:string<? r6rs:string<=? r6rs:string>=? r6rs:
    string>?)
  [(true-type-bits false-type-bits)
   (#f string-type-bits string-type-bits)
   (#f string-type-bits string-type-bits)
   (#f string-type-bits string-type-bits) ...])

(make-prim (string=? string<? string<=? string>=? string>?)
  [(true-type-bits false-type-bits)
   (#f string-type-bits string-type-bits)
   (#f string-type-bits string-type-bits) ...])

(make-prim (string->symbol)
  [(symbol-type-bits bot-type)
   (#f string-type-bits string-type-bits)])

(make-prim (substring)
  [(string-type-bits bot-type)
   (#f string-type-bits string-type-bits)
   (#f fxnum-type-bits fxnum-type-bits)
   (#f fxnum-type-bits fxnum-type-bits)])

(make-prim (string-normalize-nfd string-normalize-nfkd
            string-normalize-nfc string-normalize-nfkc)
  [(string-type-bits bot-type)
   (#f string-type-bits string-type-bits)])

(make-prim (make-string)
  [(string-type-bits bot-type)
   (#f fxnum-type-bits fxnum-type-bits)]
  [(string-type-bits bot-type)
   (#f fxnum-type-bits fxnum-type-bits)
   (#f char-type-bits char-type-bits)])

;; Can't use unknown-prim b/c that would escape it's arguments
;; for Chez we my be more precise and say void instead of top-type
(make-prim (set-car! set-cdr!) [(top-type top-type)
                                (#f pair-type-bits pair-type-bits)
                                (#t top-type top-type)])

(make-prim (length) [(length-type-bits bot-type)
                     (#f pair-or-null-type-bits pair-or-null-type-bits)])

;; vector related primitives
(make-pred-prim (vector?) vector-type-bits)

(make-top-prim (vector) [(vector-type-bits bot-type) #t ...])

(make-prim (vector->list)
  [(pair-or-null-type-bits bot-type)
   (#f vector-type-bits vector-type-bits)])

(make-prim (make-vector)
  [(vector-type-bits bot-type) (#f length-type-bits length-type-bits)]
  [(vector-type-bits bot-type) (#f length-type-bits length-type-bits)
                               (#t top-type top-type)])

(make-prim (vector-length) [(length-type-bits bot-type)
                            (#f vector-type-bits vector-type-bits)])
```

```
(make-prim (vector-ref) [(true-type false-type)
                         (#f vector-type-bits vector-type-bits)
                         (#f length-type-bits length-type-bits)])

;; for Chez we may be more precise and say void instead of top-type
(make-prim (vector-set!) [(true-type bot-type)
                          (#f vector-type-bits vector-type-bits)
                          (#f length-type-bits length-type-bits)
                          (#t top-type top-type)])

;; general integer related primitives
;;(make-pred-prim (integer?) integer-type-bits)

(make-pred-prim (bignum?) bignum-type-bits)

;; +0.0, +0.0i, etc. are also zero so we have to include other-type-bits
(make-pred-prim (zero?) (bitwise-ior fxzero-type-bits other-number-type-bits))
(make-prim (fxzero?)
  [(true-type-bits false-type-bits)
   (#f fxzero-type-bits (bitwise-xor fxnum-type-bits fxzero-type-bits))])

;; fixnum related primitives
;; TODO: if/when we "cheat" and make overflow checks explicit, these will
;;       need to have a return type of integer-type-bits and we'll rely on the
;;       check afterwards cleaning up for us.
(make-pred-prim (fixnum?) fxnum-type-bits)

(make-prim (fx+ fx* fxxor)
  [(fxnum-type-bits bot-type)
   (#f fxnum-type-bits fxnum-type-bits) ...])

(make-prim (fx- fx/)
  [(fxnum-type-bits bot-type)
   (#f fxnum-type-bits fxnum-type-bits)]
  [(fxnum-type-bits bot-type)
   (#f fxnum-type-bits fxnum-type-bits)
   (#f fxnum-type-bits fxnum-type-bits) ...])

(make-prim (fxarithmetic-shift-left fxarithmetic-shift-right fxarithmetic-shift)
  [(fxnum-type-bits bot-type)
   (#f fxnum-type-bits fxnum-type-bits)
   (#f fxnum-type-bits fxnum-type-bits)]) ; TODO: add a way to indicate
       sub-ufixnum

;;; TODO: defines prim boxes for usafe-fx<=, fx<, fx=, fx>, fx>= that will
;;;       better embody the more specific information we're learning
;;;       from them.
; fx<=
; fx<
; fx=
; fx>=
; fx>
;; TODO: why does Chez have both 'fx<' and 'fx<?'
;; TODO: why does Chez allow only one argument but not zero arguments?
(make-prim (fx<= fx< fx= fx>= fx>)
  [(true-type-bits false-type-bits)
   (#f fxnum-type-bits fxnum-type-bits)
   (#f fxnum-type-bits fxnum-type-bits) ...])
```

```
(make-prim (fx<? fx=? fx<=? fx>=? fx>?)
  [(true-type-bits false-type-bits)
   (#f fxnum-type-bits fxnum-type-bits)
   (#f fxnum-type-bits fxnum-type-bits)
   (#f fxnum-type-bits fxnum-type-bits) ...])

(make-pred-prim (flonum?) flonum-type-bits)

(make-prim (fl+ fl*) [(flonum-type-bits bot-type)
                      (#f flonum-type-bits flonum-type-bits) ...])

(make-prim (fldiv flmod fldiv0 flmod0 flexpt)
  [(flonum-type-bits bot-type)
   (#f flonum-type-bits flonum-type-bits)
   (#f flonum-type-bits flonum-type-bits)])

(make-prim (flatan fllog)
  [(flonum-type-bits bot-type)
   (#f flonum-type-bits flonum-type-bits)]
  [(flonum-type-bits bot-type)
   (#f flonum-type-bits flonum-type-bits)
   (#f flonum-type-bits flonum-type-bits)])

(make-prim (flsqrt flabs flnumerator fldenominator flfloor flceiling
            fltruncate flround flexp flsin flcos fltan flasin flacos)
  [(flonum-type-bits bot-type)
   (#f flonum-type-bits flonum-type-bits)])

(make-prim (fl- fl/ flmax flmin)
  [(flonum-type-bits bot-type)
   (#f flonum-type-bits flonum-type-bits)]
  [(flonum-type-bits bot-type)
   (#f flonum-type-bits flonum-type-bits)
   (#f flonum-type-bits flonum-type-bits) ...])

(make-prim (fl>? fl<? fl>=? fl<=? fl=?)
  [(true-type-bits false-type-bits)
   (#f flonum-type-bits flonum-type-bits)
   (#f flonum-type-bits flonum-type-bits)
   (#f flonum-type-bits flonum-type-bits) ...])

(make-prim (flpositive? flnegative? flzero? flinteger? flodd? fleven?
            flfinite? flinfinite? flnan?)
  [(true-type-bits false-type-bits)
   (#f flonum-type-bits flonum-type-bits)])

(make-pred-prim (number?) number-type-bits)

;; other type-predicate primitives
(make-pred-prim (boolean?) boolean-type-bits)

(make-pred-prim (symbol?) symbol-type-bits)

(make-pred-prim (null?) null-type-bits)

;; TODO: can't use make-pred-prim b/c that is only for non-proc-type-bits
(make-prim (procedure?)
  [(true-type-bits false-type-bits)
```

```
      (#f proc-type non-proc-type-bits)])

   ;;; TODO: defines prim box for eq? that will embody the more specific
   ;;; information we're learning from eq?
   ;;; TODO: Add to case-lambda the ability to flag it as having been used
   ;;;       in an eq? so we need to preserve it, even if we could normally
   ;;;       reduce it.

   ;; TODO: when we add ports, we'll need a display, and need to make
   ;;       the checks explicit for display
   (make-top-prim (display)
     ;; for Chez we may be more precise and say void instead of top-type
     [(true-type false-type) #f]
     [(true-type false-type) #f #f])

   ;; if void is really unspecified, it should be top here.
   (make-top-prim (void) [(other-type-bits bot-type)])

   ;; TODO: when we add ports, we'll need an read, and need to make
   ;;       the checks explicit for read
   (make-top-prim (read)
     ;; TODO: do we need to type-intersect the return values?
     [(non-proc-type-bits non-proc-type-bits)]
     [(non-proc-type-bits non-proc-type-bits) #f])

   ;; The make-bottom that always returns a bottom type
   (make-top-prim (make-bottom) [(bot-type bot-type)])

   (list (cons 'unknown-prim (lambda (ast) top-type))) ;; prims could be non-procs

   ))

(define (prim-name->type ast name)
  (let ([prim (assq name prim-table)])
    (if prim
        ((cdr prim) ast)
        (prim-name->type ast 'unknown-prim))))

)
```

B.4. Auxiliary code

Listing B.22: datatype.ss

```
(library (datatype)
  (export
   define-datatype ;; Macro (see comment for syntax)
   list-of ;; (obj -> bool) -> (obj -> bool)
   vector-of ;; (obj -> bool) -> (obj -> bool)
   improper-list-of ;; (obj -> bool) -> (obj -> bool)
   )
  (import (rnrs) (syntax-helpers) (records-io)
```

```
          (only (chezscheme) optimize-level void))

;; Disables checks at Chez Scheme optimize-level 3
(define-syntax unless-unchecked
  (lambda (x)
    (syntax-case x ()
      [(_ body ...)
       (if (fx=? (optimize-level) 3)
           #'(void)
           #'(begin body ...))])))

(define (list-of pred)
  (lambda (x*) (and (list? x*) (for-all pred x*))))

(define (vector-of pred)
  (lambda (x*) (and (vector? x*) (for-all pred (vector->list x*)))))

(define (improper-list-of pred)
  (lambda (x*)
    (or (null? x*)
        (pred x*)
        (and (pair? x*) (pred (car x*)) ((improper-list-of pred) (cdr x*))))))

(define-syntax define-case
  #;(syntax-summary
     (define-case name extract-tag
       [name tag extractor extractor]
       [name tag extractor extractor])

     (name-case expr
       [(name field field) body]
       [(name field field) body]
       [else body]))

  (lambda (stx)
    (syntax-case stx ()
      [(_ case-name tag-extractor [name tag field-extractor ...] ...)
       (syntax-for-all identifier? #'(name ...))
       (with-syntax ([((field ...) ...)
                      (syntax-map generate-temporaries
                                  #'((field-extractor ...) ...))])
         #'(define-syntax case-name
             (lambda (stx)
               (define (make-clause clause)
                 (syntax-case clause (name ...)
                   [[(name field ...) expr0 expr* (... ...)]
                    #'[(tag)
                       (let ([field (field-extractor expr-var)] ...)
                         #f ;; force non-definition context like scheme 'case'
                         expr0 expr* (... ...))]] ...))
               (syntax-case stx (else)
                 [(_ expr
                     [(clause-name field-name (... ...)) body0 body* (... ...)]
                     (... ...))
                  (syntax-for-all (lambda (x) (syntax-for-all identifier? x))
                                  #'((field-name (... ...)) (... ...)))
                  ;; TODO: check for missing clause
                  #'(case-name expr
                      [(clause-name field-name (... ...)) body0 body* (... ...)]
```

```
                    (... ...)
                    [else (assertion-violation 'case-name "missing case")])]
                [(_ expr
                    [(clause-name field-name (... ...)) body0 body* (... ...)]
                    (... ...)
                    [else else-body0 else-body* (... ...)])
                 (syntax-for-all (lambda (x) (syntax-for-all identifier? x))
                                 #'((field-name (... ...)) (... ...)))
                 ;; TODO: check for duplicate clause
                 #`(let ([expr-var expr])
                     (case (tag-extractor expr-var)
                       #,@(syntax-map
                           make-clause
                           #'([(clause-name field-name (... ...))
                               body0 body* (... ...)] (... ...)))
                       [else else-body0 else-body* (... ...)])))])))])))

(define-syntax define-datatype
  #;(syntax-summary
     (define-datatype (AST (property default type) (property default))
       [Constor (field type) field]
       [Constor field (field type)]))

  (lambda (stx)
    (define (normalize-property-spec stx)
      (syntax-case stx ()
        [(property default) (identifier? #'property)
         #'(property default (lambda (x) #t))]
        [(property default predicate) (identifier? #'property)
         #'(property default predicate)]))
    (define (normalize-field-spec stx)
      (syntax-case stx ()
        [field (identifier? #'field) #'(field (lambda (x) #t))]
        [(field predicate) (identifier? #'field)  #'(field predicate)]))
    (syntax-case stx ()
      [(_ (datatype property-spec ...)
          [constructor field-spec ...] ...)
       ;; exports:
       ;;  datatype(record) datatype?
       ;;  datatype-property datatype-property-set!
       ;;  datatype-case constructor constructor?
       (and (identifier? #'datatype) (for-all identifier? #'(constructor ...)))

       ;; expand property-spec and field-spec
       (with-syntax ([((property property-default property-predicate) ...)
                      (syntax-map normalize-property-spec #'(property-spec ...))]
                     [(((field field-predicate) ...) ...)
                      (syntax-map
                       (lambda (x) (syntax-map normalize-field-spec x))
                       #'((field-spec ...) ...))])

         ;; public exports
         (with-syntax ([datatype? (syntax-append #'datatype #'datatype "?")]
                       [datatype-case (syntax-append #'datatype #'datatype "-case")]
                       [(datatype-property ...)
                        (syntax-map
                         (lambda (x) (syntax-append #'datatype #'datatype "-" x))
                         #'(property ...))]
                       [(datatype-property-set! ...)
```

```
                    (syntax-map
                     (lambda (x)
                       (syntax-append #'datatype #'datatype "-" x "-set!"))
                     #'(property ...))])

          ;; private identifiers
          (with-syntax ([(datatype-property-unchecked-set! ...)
                         (syntax-map
                          (lambda (x)
                            (syntax-append
                             #'private #'datatype "-" x "-unchecked-set!"))
                          #'(property ...))]
                        [((constructor-field ...) ...)
                         (syntax-map
                          (lambda (constructor fields)
                            (syntax-map
                             (lambda (field)
                               (syntax-append #'private constructor "-" field))
                             fields))
                          #'(constructor ...)
                          #'((field ...) ...))]
                        [(constructor-record ...)
                         (syntax-map (lambda (x)
                                       (syntax-append #'private x "-record"))
                                     #'(constructor ...))]
                        [(constructor? ...)
                         (syntax-map (lambda (x) (syntax-append #'private x "?"))
                                     #'(constructor ...))]
                        [(constructor-number ...)
                         (syntax-indexes #'(constructor ...))])

            ;; resulting code
            #'(begin
                ;; Define parent record
                ;;   private: make-datatype and datatype-variant
                (define-record-type (datatype make-datatype datatype?)
                  (fields (immutable variant datatype-variant)
                          (mutable property datatype-property
                                   datatype-property-unchecked-set!) ...)
                  (nongenerative)
                  (protocol (lambda (make)
                              (lambda (variant)
                                (make variant property-default ...)))))
                (define (datatype-property-set! obj value)
                  (unless-unchecked
                   (unless (property-predicate value)
                     (assertion-violation 'datatype-property-set!
                                          "invalid value" value)))
                  (datatype-property-unchecked-set! obj value))
                ...

                ;; Define child records
                (define-record-type
                  (constructor-record constructor constructor?)
                  (parent datatype)
                  (fields (immutable field constructor-field) ...)
                  (sealed #t)
                  (nongenerative)
                  (protocol
```

```
                    (lambda (parent)
                      (lambda (field ...)
                        (unless-unchecked
                         (define who 'constructor)
                         (unless (field-predicate field)
                           (assertion-violation
                            who "invalid value for field" 'field field))
                         ...)
                        ((parent constructor-number) field ...)))))
                ...

                ;; Define case form
                (define-case datatype-case datatype-variant
                  [constructor constructor-number constructor-field ...] ...)

                ;; Define reader/writer
                (define *ignored* ;; maintain definition context
                  (begin
                    (record-reader
                     'constructor (record-type-descriptor constructor-record))
                    ...
                    (record-writer (record-type-descriptor constructor-record)
                      (lambda (x p wr)
                        (wr `(constructor
                              ,@(map (lambda (a) (a x))
                                     (list constructor-field ...))) p)))
                    ...))
                ))))]))))
)
```

Listing B.23: define-ignored.ss

```
(library (define-ignored)
  (export define-ignored)
  (import (rnrs))

(define-syntax define-ignored
  (syntax-rules ()
    [(_ expr)
     (define *ignored* (let-values ([ignored expr]) #f))]))

)
```

Listing B.24: iota.ss

```
(library (iota)
  (export iota)
  (import (rnrs))

;; n -> '(0 1 2 3 ... n-1)
(define (iota x) (reverse ($iota x)))

(define ($iota x)
  (cond
   [(zero? x) '()]
```

```
   [else (cons (- x 1) ($iota (- x 1)))]))

)
```

Listing B.25: records-io.ss

```
(library (records-io)
  (export record-writer record-reader)
  (import (only (chezscheme) record-writer record-reader)))
```

Listing B.26: safe-forms.ss

```
(library (safe-forms)
  (export if cond case)
  (import (rename (rnrs) (if rnrs:if) (cond rnrs:cond) (case rnrs:case)))

;; Versions of 'if', 'cond', and 'case' that are slightly safer to use:
;;   'if' - rejects one-armed if
;;   'cond' - errors if no cases match
;;   'case' - errors if no cases match, also allows a bare symbol
;; There are modified from R6RS - Appendix B

(define-syntax if
  (lambda (x)
    (syntax-case x ()
      [(_ t c) (raise (syntax-violation 'if "found␣one-armed␣if" x))]
      [(_ t c a) #'(rnrs:if t c a)])))

(define-syntax cond
  (syntax-rules (else =>)
    ((cond (else result1 result2 ...))
     (begin result1 result2 ...))
    ((cond (test => result))
     (let ((temp test))
       (if temp (result temp) (assertion-violation 'cond "no␣clauses␣match"))))
    ((cond (test => result) clause1 clause2 ...)
     (let ((temp test))
       (if temp
           (result temp)
           (cond clause1 clause2 ...))))
    ((cond (test)) (or test (assertion-violation 'cond "no␣clauses␣match")))
    ((cond (test) clause1 clause2 ...)
     (let ((temp test))
       (if temp
           temp
           (cond clause1 clause2 ...))))
    ((cond (test result1 result2 ...))
     (if test (begin result1 result2 ...)
         (assertion-violation 'cond "no␣clauses␣match")))
    ((cond (test result1 result2 ...)
           clause1 clause2 ...)
     (if test
         (begin result1 result2 ...)
         (cond clause1 clause2 ...)))))
```

```
(define-syntax case
  (syntax-rules (else)
    [(case e
       [s* r* r** ...]
       ...
       [else else-r else-r* ...])
     (let ([t e])
       (cond
         [(case-test t s*) r* r** ...]
         ...
         [else else-r else-r* ...]))]
    [(case e
       [s r1 r1* ...]
       [s* r2* r2** ...]
       ...)
     (let ([t e])
       (cond
         [(case-test t s) r1 r1* ...]
         [(case-test t s*) r2* r2** ...]
         ...
         [else (assertion-violation 'case "no clauses match")]))]))

(define-syntax case-test
  (syntax-rules ()
    [(_ v (m* ...)) (memv v '(m* ...))]
    [(_ v m) (eqv? v 'm)]))

)
```

Listing B.27: syntax-helpers.ss

```
(library (syntax-helpers)
  (export
   syntax-ellipsis? ;; syntax -> bool
   syntax-indexes ;; #'(a b c ...) -> #'((1 . a) (2 . b) (3 . c) ...)
   syntax-length ;; syntax -> int
   syntax-car syntax-cdr ;; syntax -> syntax
   syntax-map syntax-for-all
   syntax-append)
  (import (rnrs))

(define (syntax-ellipsis? stx)
  (and (identifier? stx) (free-identifier=? stx #'(... ...))))

(define syntax-indexes
  (case-lambda
   [(list) (syntax-indexes list 0)]
   [(list base)
    (syntax-case list ()
      [() #'()]
      [(x . y) (with-syntax ([base base]
                             [y (syntax-indexes #'y (+ base 1))])
                 #'((base . y))])])))

(define (syntax-length stx)
  (syntax-case stx ()
    [() 0]
```

```
    [(car . cdr) (+ 1 (syntax-length #'cdr))]))

(define (syntax-car stx) (syntax-case stx () [(car . cdr) #'car]))
(define (syntax-cdr stx) (syntax-case stx () [(car . cdr) #'cdr]))

(define (syntax-map function . stx)
  (syntax-case stx ()
    [(() ...) #'()]
    [else
     #`(#,(apply function (map syntax-car stx)) .
        #,(apply syntax-map function (map syntax-cdr stx)))]))

(define (syntax-for-all function . stx)
  (syntax-case stx ()
    [(() ...) #t]
    [else
     (and (apply function (map syntax-car stx))
          (apply syntax-for-all function (map syntax-cdr stx)))]))

(define (syntax-append template-identifier . args)
  (datum->syntax
   template-identifier
   (string->symbol
    (apply string-append
           (map (lambda (x)
                  (cond
                   [(string? x) x]
                   [(symbol? x) (symbol->string x)]
                   [(identifier? x) (symbol->string (syntax->datum x))]))
                args)))))

)
```

B.5. Third-party Code

B.5.1. SRFI 39

The following code is based on code in Dybvig [2010] and is used by permission.

Listing B.28: srfi-39.ss

```
(library (srfi-39)
  (export make-parameter parameterize)
  (import (rnrs))

(define make-parameter
  (case-lambda
   [(init) (make-parameter init (lambda (x) x))]
   [(init converter)
    ;; TODO: We simulate a parameter here.  For thread safety this
    ;; should become define-threaded which uses a threaded-parameter.
```

```
    (let ([value (converter init)])
      (let ([parameter ;; give the lambda a name
             (case-lambda
              [() value]
              [(new-value) (set! value (converter new-value))])])
        parameter))))

;; WARNING: converters must be idempotent
(define-syntax parameterize
  (lambda (stx)
    (syntax-case stx ()
      [(_ ([lhs rhs] ...) body body* ...)
       (with-syntax ([(param ...) (generate-temporaries #'(lhs ...))]
                     [(local ...) (generate-temporaries #'(rhs ...))])
         #'(let ([param lhs] ...
                 [local rhs] ...)
             (let ([swap (lambda ()
                           (let ([tmp (param)])
                             (param local)
                             (set! local tmp)) ...)])
               (dynamic-wind swap (lambda () body body* ...) swap))))])))

)
```

B.5.2. Match

The following code is based on code at:

`http://www.cs.indiana.edu/chezscheme/match/`

The version included here has been modified and ported to be an R6RS library.

Listing B.29: match.ss

```
#!chezscheme
(library (match)
  (export match)
  (import (scheme))

;;; Copyright (c) 2000-2008 Dan Friedman, Erik Hilsdale, and Kent Dybvig
;;;
;;; Permission is hereby granted, free of charge, to any person
;;; obtaining a copy of this software and associated documentation files
;;; (the "Software"), to deal in the Software without restriction,
;;; including without limitation the rights to use, copy, modify, merge,
;;; publish, distribute, sublicense, and/or sell copies of the Software,
;;; and to permit persons to whom the Software is furnished to do so,
;;; subject to the following conditions:
;;;
;;; The above copyright notice and this permission notice shall be
;;; included in all copies or substantial portions of the Software.
;;;
;;; THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND,
```

```
;;; EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF
;;; MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND
;;; NONINFRINGEMENT.  IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS
;;; BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN
;;; ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN
;;; CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE
;;; SOFTWARE.

;;; This program was originally designed and implemented by Dan Friedman.
;;; It was redesigned and reimplemented by Erik Hilsdale.  Additional
;;; modifications were made by Kent Dybvig, Steve Ganz, and Aziz Ghuloum.
;;; Parts of the implementation were adapted from the portable syntax-case
;;; implementation written by Kent Dybvig, Oscar Waddell, Bob Hieb, and
;;; Carl Bruggeman and is used by permission of Cadence Research Systems.

;;; A change log appears at end of this file.

;;; A brief description of match is given at:

;;;   http://www.cs.indiana.edu/chezscheme/match/

;;; ============================================================

;; Exp     ::= (match               Exp Clause)
;;           || (trace-match         Exp Clause)
;;           || (match+       (Id*) Exp Clause*)
;;           || (trace-match+ (Id*) Exp Clause*)
;;           || OtherSchemeExp

;; Clause ::= (Pat Exp+) || (Pat (guard Exp*) Exp+)

;; Pat     ::= (Pat ... . Pat)
;;           || (Pat . Pat)
;;           || ()
;;           || #(Pat* Pat ... Pat*)
;;           || #(Pat*)
;;           || ,Id
;;           || ,[Id*]
;;           || ,[Cata -> Id*]
;;           || Id

;; Cata    ::= Exp

;; YOU'RE NOT ALLOWED TO REFER TO CATA VARS IN GUARDS. (reasonable!)

(module ((match+ match-help match-help1 clause-body let-values**
           guard-body convert-pat mapper my-backquote extend-backquote
           sexp-dispatch)
         (trace-match+ match-help match-help1 clause-body let-values**
           guard-body convert-pat mapper my-backquote extend-backquote
           sexp-dispatch)
         (match match-help match-help1 clause-body let-values**
           guard-body convert-pat mapper my-backquote extend-backquote
           sexp-dispatch)
         (trace-match match-help match-help1 clause-body let-values**
           guard-body convert-pat mapper my-backquote extend-backquote
           sexp-dispatch)
         (with-ellipsis-aware-quasiquote my-backquote)
         match-equality-test)
```

```
(import scheme)

(define match-equality-test
  (make-parameter
    equal?
    (lambda (x)
      (unless (procedure? x)
        (error 'match-equality-test "~s␣is␣not␣a␣procedure" x))
      x)))

(define-syntax match+
  (lambda (x)
    (syntax-case x ()
      [(k (ThreadedId ...) Exp Clause ...)
       #'(let f ((ThreadedId ThreadedId) ... (x Exp))
           (match-help k f x (ThreadedId ...) Clause ...))])))

(define-syntax match
  (lambda (x)
    (syntax-case x ()
      [(k Exp Clause ...)
       #'(let f ((x Exp))
           (match-help k f x () Clause ...))])))

(define-syntax trace-match+
  (lambda (x)
    (syntax-case x ()
      [(k (ThreadedId ...) Name Exp Clause ...)
       #'(letrec ((f (trace-lambda Name (ThreadedId ... x)
                       (match-help k f x (ThreadedId ...) Clause ...))))
           (f ThreadedId ... x))])))

(define-syntax trace-match
  (lambda (x)
    (syntax-case x ()
      [(k Name Exp Clause ...)
       #'(letrec ((f (trace-lambda Name (x)
                       (match-help k f x () Clause ...))))
           (f Exp))])))

;;; ----------------------------

(define-syntax let-values**
  (syntax-rules ()
    ((_ () B0 B ...) (begin B0 B ...))
    ((_ ((Formals Exp) Rest ...) B0 B ...)
     (let-values** (Rest ...)
       (call-with-values (lambda () Exp)
         (lambda Formals B0 B ...))))))

(define-syntax match-help
  (lambda (x)
    (syntax-case x ()
      ((_ Template Cata Obj ThreadedIds)
       #'(error 'match "Unmatched␣datum:␣~s" Obj))
      ((_ Template Cata Obj ThreadedIds (Pat B0 B ...) Rest ...)
       #'(convert-pat Pat
           (match-help1 Template Cata Obj ThreadedIds
```

```
             (B0 B ...)
             Rest ...)))
      ((_ Template Cata Obj ThreadedIds cls Rest ...)
       (syntax-error #'cls "invalid␣match␣clause")))))


(define-syntax match-help1
  (lambda (x)
    (syntax-case x (guard)
      [(_ PatLit Vars () Cdecls Template Cata Obj ThreadedIds
         ((guard) B0 B ...) Rest ...)
       #'(let ((ls/false (sexp-dispatch Obj PatLit)))
           (if ls/false
               (apply (lambda Vars
                        (clause-body Cata Cdecls ThreadedIds
                          (extend-backquote Template B0 B ...)))
                 ls/false)
               (match-help Template Cata Obj ThreadedIds Rest ...)))]
      [(_ PatLit Vars (PG ...) Cdecls Template Cata Obj ThreadedIds
         ((guard G ...) B0 B ...) Rest ...)
       #'(let ((ls/false (sexp-dispatch Obj PatLit)))
           (if (and ls/false (apply (lambda Vars
                                      (guard-body Cdecls
                                        (extend-backquote Template
                                          (and PG ... G ...))))
                                    ls/false))
               (apply (lambda Vars
                        (clause-body Cata Cdecls ThreadedIds
                          (extend-backquote Template B0 B ...)))
                 ls/false)
               (match-help Template Cata Obj ThreadedIds Rest ...)))]
      [(_ PatLit Vars (PG ...) Cdecls Template Cata Obj ThreadedIds
         (B0 B ...) Rest ...)
       #'(match-help1 PatLit Vars (PG ...) Cdecls Template Cata Obj ThreadedIds
           ((guard) B0 B ...) Rest ...)])))

(define-syntax clause-body
  (lambda (x)
    (define build-mapper
      (lambda (vars depth cata tIds)
        (if (zero? depth)
            cata
            (with-syntax ((rest (build-mapper vars (- depth 1) cata tIds))
                          (vars vars)
                          (tIds tIds))
              #'(mapper rest vars tIds)))))
    (syntax-case x ()
      ((_ Cata ((CVar CDepth CMyCata CFormal ...) ...) (ThreadedId ...) B)
       (with-syntax (((Mapper ...)
                      (map (lambda (mycata formals depth)
                             (build-mapper formals
                               (syntax->datum depth)
                               (syntax-case mycata ()
                                 [#f #'Cata]
                                 [exp #'exp])
                               #'(ThreadedId ...)))
                        #'(CMyCata ...)
                        #'((CFormal ...) ...)
                        #'(CDepth ...))))
```

```
        #'(let-values** (([ThreadedId ... CFormal ...]
                         (Mapper ThreadedId ... CVar))
                        ...)
            B))))))

(define-syntax guard-body
  (lambda (x)
    (syntax-case x ()
      ((_ ((Cvar Cdepth MyCata Cformal ...) ...) B)
       (with-syntax (((CF ...) (apply append #'((Cformal ...) ...))))
         #'(let-syntax
               ((CF
                  (lambda (x)
                    (syntax-case x ()
                      (Name
                        (syntax-error #'Name
                          "guard␣cannot␣refer␣to␣return-value␣variable")))))
                ...)
             B))))))

(define-syntax convert-pat
  ;; returns sexp-pat x vars x guards x cdecls
  (let ()
    (define ellipsis?
      (lambda (x)
        (and (identifier? x) (free-identifier=? x #'(... ...)))))
    (define Var?
      (lambda (x)
        (syntax-case x (->)
          [-> #f]
          [id (identifier? #'id)])))
    (define fVar
      (lambda (var vars guards)
        (let loop ([ls vars])
          (if (null? ls)
              (values (cons var vars) guards)
              (if (bound-identifier=? var (car ls))
                  (with-syntax ([(tmp) (generate-temporaries (list var))]
                                [var (car ls)])
                    (values (cons #'tmp vars)
                            (cons #'((match-equality-test) tmp var) guards)))
                  (loop (cdr ls))))))))
    (define (f syn vars guards cdecls depth)
      (syntax-case syn (unquote)
        ((unquote . stuff) ; separate for better error detection
         (syntax-case syn (unquote ->)
           ((unquote [MyCata -> Var ...])
            (andmap Var? #'(Var ...))
            (with-syntax (((Temp) (generate-temporaries '(x)))
                          (Depth depth))
              (values #'any
                      (cons #'Temp vars)
                      guards
                      (cons #'[Temp Depth MyCata Var ...] cdecls))))
           ((unquote [Var ...])
            (andmap Var? #'(Var ...))
            (with-syntax (((Temp) (generate-temporaries '(x)))
                          (Depth depth))
              (values #'any
```

```
                (cons #'Temp vars)
                guards
                (cons #'[Temp Depth #f Var ...] cdecls))))
    ((unquote Var)
     (Var? #'Var)
     (let-synvalues* ([(vars guards) (fVar #'Var vars guards)])
       (values #'any #'vars #'guards cdecls)))))
  (((unquote . stuff) Dots)
   (ellipsis? #'Dots)
   (syntax-case syn (unquote ->)
     (((unquote [MyCata -> Var ...]) Dots)
      (andmap Var? #'(Var ...))
      (with-syntax (((Temp) (generate-temporaries '(x)))
                    (Depth+1 (add1 depth)))
        (values #'each-any
                (cons #'Temp vars)
                guards
                (cons #'[Temp Depth+1 MyCata Var ...] cdecls))))
     (((unquote [Var ...]) Dots)
      (andmap Var? #'(Var ...))
      (with-syntax (((Temp) (generate-temporaries '(x)))
                    (Depth+1 (add1 depth)))
        (values #'each-any
                (cons #'Temp vars)
                guards
                (cons #'[Temp Depth+1 #f Var ...] cdecls))))
     (((unquote Var) Dots)
      (Var? #'Var)
      (let-synvalues* ([(vars guards) (fVar #'Var vars guards)])
        (values #'each-any #'vars #'guards cdecls)))
     ((expr Dots) (syntax-error #'expr "match-pattern␣unquote␣syntax"))))
  ((Pat Dots)
   (ellipsis? #'Dots)
   (let-synvalues* (((Dpat Dvars Dguards Dcdecls)
                     (f #'Pat vars guards cdecls (add1 depth))))
     (with-syntax ((Size (- (length #'Dvars) (length vars))))
       (values #'#(each Dpat Size) #'Dvars #'Dguards #'Dcdecls))))
  ((Pat Dots . Rest)
   (ellipsis? #'Dots)
   (let-synvalues* (((Rpat Rvars Rguards Rcdecls)
                     (f #'Rest vars guards cdecls depth))
                    ((Dpat Dvars Dguards Dcdecls)
                     (f #'(Pat (... ...)) #'Rvars #'Rguards #'Rcdecls
                       depth)))
     (with-syntax ((Size (- (length #'Dvars) (length #'Rvars)))
                   ((RevRestTl . RevRest) (reverseX #'Rpat '())))
       (values #'#(tail-each Dpat Size RevRest RevRestTl)
               #'Dvars #'Dguards #'Dcdecls))))
  ((X . Y)
   (let-synvalues* (((Ypat Yvars Yguards Ycdecls)
                     (f #'Y vars guards cdecls depth))
                    ((Xpat Xvars Xguards Xcdecls)
                     (f #'X #'Yvars #'Yguards #'Ycdecls depth)))
     (values #'(Xpat . Ypat) #'Xvars #'Xguards #'Xcdecls)))
  (() (values #'() vars guards cdecls))
  (#(X ...)
   (let-synvalues* (((Pat Vars Eqvars Cdecls)
                     (f #'(X ...) vars guards cdecls depth)))
     (values #'#(vector Pat) #'Vars #'Eqvars #'Cdecls)))
```

```
        (Thing (values #'#(atom Thing) vars guards cdecls))))
    (define reverseX
      (lambda (ls acc)
        (if (pair? ls)
            (reverseX (cdr ls) (cons (car ls) acc))
            (cons ls acc))))
    (define-syntax let-synvalues*
      (syntax-rules ()
        ((_ () B0 B ...) (begin B0 B ...))
        ((_ (((Formal ...) Exp) Decl ...) B0 B ...)
         (call-with-values (lambda () Exp)
           (lambda (Formal ...)
             (with-syntax ((Formal Formal) ...)
               (let-synvalues* (Decl ...) B0 B ...)))))))
    (lambda (syn)
      (syntax-case syn ()
        ((_ syn (kh . kt))
         (let-synvalues* (((Pat Vars Guards Cdecls) (f #'syn '() '() '() 0)))
           #'(kh 'Pat Vars Guards Cdecls . kt)))))))

(define-syntax mapper
  (lambda (x)
    (syntax-case x ()
      ((_ F (RetId ...) (ThreadId ...))
       (with-syntax (((t ...) (generate-temporaries #'(RetId ...)))
                     ((ts ...) (generate-temporaries #'(RetId ...)))
                     ((null ...) (map (lambda (x) #''()) #'(RetId ...))))
         #'(let ((fun F))
             (rec g
               (lambda (ThreadId ... ls)
                 (if (null? ls)
                     (values ThreadId ... null ...)
                     (call-with-values
                         (lambda () (g ThreadId ... (cdr ls)))
                       (lambda (ThreadId ... ts ...)
                         (call-with-values
                             (lambda () (fun ThreadId ... (car ls)))
                           (lambda (ThreadId ... t ...)
                             (values ThreadId ... (cons t ts) ...)))))))))))))

;;; ----------------------------

(define-syntax my-backquote
  (lambda (x)
    (define ellipsis?
      (lambda (x)
        (and (identifier? x) (free-identifier=? x #'(... ...)))))
    (define-syntax with-values
      (syntax-rules ()
        ((_ P C) (call-with-values (lambda () P) C))))
    (define-syntax syntax-lambda
      (lambda (x)
        (syntax-case x ()
          ((_ (Pat ...) Body0 Body ...)
           (with-syntax (((X ...) (generate-temporaries #'(Pat ...))))
             #'(lambda (X ...)
                 (with-syntax ((Pat X) ...)
                   Body0 Body ...)))))))
    (define-syntax with-temp
```

```
  (syntax-rules ()
    ((_ V Body0 Body ...)
     (with-syntax (((V) (generate-temporaries '(x))))
       Body0 Body ...))))
(define-syntax with-temps
  (syntax-rules ()
    ((_ (V ...) (Exp ...) Body0 Body ...)
     (with-syntax (((V ...) (generate-temporaries #'(Exp ...))))
       Body0 Body ...))))
(define destruct
  (lambda (Orig x depth)
    (syntax-case x (quasiquote unquote unquote-splicing)
      ;; inner quasiquote
      ((Exp dots1 dots2 . Rest)
       (and (zero? depth) (ellipsis? #'dots1) (ellipsis? #'dots2))
       (let f ([Exp #'(... ((Exp ...) ...))] [Rest #'Rest] [ndots 2])
         (syntax-case Rest ()
           [(dots . Rest)
            (ellipsis? #'dots)
            (with-syntax ([Exp Exp])
              (f #'(... (Exp ...)) #'Rest (+ ndots 1)))]
           [Rest
            (with-values (destruct Orig Exp depth)
              (syntax-lambda (ExpBuilder (ExpVar ...) (ExpExp ...))
                (if (null? #'(ExpVar ...))
                    (syntax-error Orig "Bad ellipsis")
                    (with-values (destruct Orig #'Rest depth)
                      (syntax-lambda (RestBuilder RestVars RestExps)
                        (values
                          #`(append
                              #,(let f ([ndots ndots])
                                  (if (= ndots 1)
                                      #'ExpBuilder
                                      #`(apply append #,(f (- ndots 1)))))
                              RestBuilder)
                          (append #'(ExpVar ...) #'RestVars)
                          (append #'(ExpExp ...) #'RestExps)))))))])))
      ((quasiquote Exp)
       (with-values (destruct Orig #'Exp (add1 depth))
         (syntax-lambda (Builder Vars Exps)
           (if (null? #'Vars)
               (values #''(quasiquote Exp) '() '())
               (values #'(list 'quasiquote Builder) #'Vars #'Exps)))))
      ;; unquote
      ((unquote Exp)
       (zero? depth)
       (with-temp X
         (values #'X (list #'X) (list #'Exp))))
      ((unquote Exp)
       (with-values (destruct Orig #'Exp (sub1 depth))
         (syntax-lambda (Builder Vars Exps)
           (if (null? #'Vars)
               (values #''(unquote Exp) '() '())
               (values #'(list 'unquote Builder) #'Vars #'Exps)))))
      ;; splicing
      (((unquote-splicing Exp))
       (zero? depth)
       (with-temp X
         (values #'X (list #'X) (list #'Exp))))
```

```
          (((unquote-splicing Exp ...))
           (zero? depth)
           (with-temps (X ...) (Exp ...)
             (values #'(append X ...) #'(X ...) #'(Exp ...))))
          (((unquote-splicing Exp ...) . Rest)
           (zero? depth)
           (with-values (destruct Orig #'Rest depth)
             (syntax-lambda (Builder Vars Exps)
               (with-temps (X ...) (Exp ...)
                 (if (null? #'Vars)
                     (values #'(append X ... 'Rest)
                             #'(X ...) #'(Exp ...))
                     (values #'(append X ... Builder)
                             #'(X ... . Vars) #'(Exp ... . Exps)))))))
          ((unquote-splicing Exp ...)
           (with-values (destruct Orig #'(Exp ...) (sub1 depth))
             (syntax-lambda (Builder Vars Exps)
               (if (null? #'Vars)
                   (values #''(unquote-splicing Exp ...) '() '())
                   (values #'(cons 'unquote-splicing Builder)
                           #'Vars #'Exps)))))
          ;; dots
          (((unquote Exp) Dots)
           (and (zero? depth) (ellipsis? #'Dots))
           (with-temp X
             (values #'X (list #'X) (list #'Exp))))
          (((unquote Exp) Dots . Rest)
           (and (zero? depth) (ellipsis? #'Dots))
           (with-values (destruct Orig #'Rest depth)
             (syntax-lambda (RestBuilder RestVars RestExps)
               (with-syntax ((TailExp
                              (if (null? #'RestVars)
                                  #''Rest
                                  #'RestBuilder)))
                 (with-temp X
                   (values #'(append X TailExp)
                           (cons #'X #'RestVars)
                           (cons #'Exp #'RestExps)))))))
          ((Exp Dots . Rest)
           (and (zero? depth) (ellipsis? #'Dots))
           (with-values (destruct Orig #'Exp depth)
             (syntax-lambda (ExpBuilder (ExpVar ...) (ExpExp ...))
               (if (null? #'(ExpVar ...))
                   (syntax-error Orig "Bad ellipsis")
                   (with-values (destruct Orig #'Rest depth)
                     (syntax-lambda (RestBuilder RestVars RestExps)
                       (with-syntax ((TailExp
                                      (if (null? #'RestVars)
                                          #''Rest
                                          #'RestBuilder))
                                     (Orig Orig))
                         (values #'(let f ((ExpVar ExpVar) ...)
                                     (if (and (pair? ExpVar) ...)
                                         (cons
                                           (let ((ExpVar (car ExpVar)) ...)
                                             ExpBuilder)
                                           (f (cdr ExpVar) ...))
                                         (if (and (null? ExpVar) ...)
                                             TailExp
```

```
                                          (error 'unquote
                                            "Mismatched␣lists␣in␣~s"
                                            Orig))))
                                  (append #'(ExpVar ...) #'RestVars)
                                  (append #'(ExpExp ...) #'RestExps))))))))
          ;; Vectors
          (#(X ...)
           (with-values (destruct Orig #'(X ...) depth)
             (syntax-lambda (LsBuilder LsVars LsExps)
               (values #'(list->vector LsBuilder) #'LsVars #'LsExps))))
          ;; random stuff
          ((Hd . Tl)
           (with-values (destruct Orig #'Hd depth)
             (syntax-lambda (HdBuilder HdVars HdExps)
               (with-values (destruct Orig #'Tl depth)
                 (syntax-lambda (TlBuilder TlVars TlExps)
                   (with-syntax ((Hd (if (null? #'HdVars)
                                         #''Hd
                                         #'HdBuilder))
                                 (Tl (if (null? #'TlVars)
                                         #''Tl
                                         #'TlBuilder)))
                     (values #'(cons Hd Tl)
                             (append #'HdVars #'TlVars)
                             (append #'HdExps #'TlExps))))))))
          (OtherThing
            (values #''OtherThing '() '()))))))
    ;; macro begins
    (syntax-case x ()
      ((_ Datum)
       (with-values (destruct #'(quasiquote Datum) #'Datum 0)
         (syntax-lambda (Builder (Var ...) (Exp ...))
           (if (null? #'(Var ...))
               #''Datum
               #'(let ((Var Exp) ...)
                   Builder)))))))

(define-syntax extend-backquote
  (lambda (x)
    (syntax-case x ()
      [(_ Template Exp ...)
       (with-syntax ([quasiquote (datum->syntax #'Template 'quasiquote)])
         #'(let-syntax ([quasiquote
                         (lambda (x)
                           (syntax-case x ()
                             ((_ Foo) #'(my-backquote Foo))))])
             Exp ...))])))

(define-syntax with-ellipsis-aware-quasiquote
  (lambda (x)
    (syntax-case x ()
      [(k b1 b2 ...)
       (with-implicit (k quasiquote)
         #'(let-syntax ([quasiquote
                         (lambda (x)
                           (syntax-case x ()
                             ((_ e) #'(my-backquote e))))])
             (let () b1 b2 ...)))])))
```

```
;;; ------------------------------

(define-syntax with-values
  (syntax-rules ()
    ((_ P C) (call-with-values (lambda () P) C))))

(define-syntax letcc
  (syntax-rules ()
    ((_ V B0 B ...) (call/cc (lambda (V) B0 B ...)))))

(define classify-list
  (lambda (ls)
    (cond
      ((null? ls) 'proper)
      ((not (pair? ls)) 'improper)
      (else
        (let f ((tortoise ls) (hare (cdr ls)))
          (cond
            ((eq? tortoise hare) 'infinite)
            ((null? hare) 'proper)
            ((not (pair? hare)) 'improper)
            (else
              (let ((hare (cdr hare)))
                (cond
                  ((null? hare) 'proper)
                  ((not (pair? hare)) 'improper)
                  (else (f (cdr ls) (cdr hare))))))))))))

(define ilist-copy-flat
  (lambda (ils)
    (let f ((tortoise ils) (hare (cdr ils)))
      (if (eq? tortoise hare)
          (list (car tortoise))
          (cons (car tortoise) (f (cdr tortoise) (cddr hare)))))))

(define sexp-dispatch
  (lambda (obj pat);; #f or list of vars
    (letcc escape
      (let ((fail (lambda () (escape #f))))
        (let f ((pat pat) (obj obj) (vals '()))
          (cond
            ((eq? pat 'any)
             (cons obj vals))
            ((eq? pat 'each-any)
             ;; handle infinities
             (case (classify-list obj)
               ((proper infinite) (cons obj vals))
               ((improper) (fail))))
            ((pair? pat)
             (if (pair? obj)
                 (f (car pat) (car obj) (f (cdr pat) (cdr obj) vals))
                 (fail)))
            ((vector? pat)
             (case (vector-ref pat 0)
               ((atom)
                (let ((a (vector-ref pat 1)))
                  (if (eqv? obj a)
                      vals
                      (fail))))
```

```
                ((vector)
                 (if (vector? obj)
                     (let ((vec-pat (vector-ref pat 1)))
                       (f vec-pat (vector->list obj) vals))
                     (fail)))
                ((each)
                 ;; if infinite, copy the list as flat, then do the matching,
                 ;; then do some set-cdrs.
                 (let ((each-pat (vector-ref pat 1))
                       (each-size (vector-ref pat 2)))
                   (case (classify-list obj)
                     ((improper) (fail))
                     ((infinite)
                      (let ((each-vals (f pat (ilist-copy-flat obj) '())))
                        (for-each (lambda (x) (set-cdr! (last-pair x) x))
                          each-vals)
                        (append each-vals vals)))
                     ((proper)
                      (append
                        (let g ((obj obj))
                          (if (null? obj)
                              (make-list each-size '())
                              (let ((hd-vals (f each-pat (car obj) '()))
                                    (tl-vals (g (cdr obj))))
                                (map cons hd-vals tl-vals))))
                        vals)))))
                ((tail-each)
                 (let ((each-pat (vector-ref pat 1))
                       (each-size (vector-ref pat 2))
                       (revtail-pat (vector-ref pat 3))
                       (revtail-tail-pat (vector-ref pat 4)))
                   (when (eq? (classify-list obj) 'infinite) (fail))
                   (with-values
                       (let g ((obj obj))
                         ;; in-tail?, vals, revtail-left/ls
                         (cond
                           ((pair? obj)
                            (with-values (g (cdr obj))
                              (lambda (in-tail? vals tail-left/ls)
                                (if in-tail?
                                    (if (null? tail-left/ls)
                                        (values #f vals (list (car obj)))
                                        (values #t (f (car tail-left/ls)
                                                      (car obj)
                                                      vals)
                                                   (cdr tail-left/ls)))
                                    (values #f vals
                                            (cons (car obj) tail-left/ls))))))
                           (else
                             (values #t
                                     (f revtail-tail-pat obj vals)
                                     revtail-pat))))
                       (lambda (in-tail? vals tail-left/ls)
                         (if in-tail?
                             (if (null? tail-left/ls)
                                 (append (make-list each-size '())
                                   vals)
                                 (fail))
                             (f each-pat tail-left/ls vals)))))))))
```

```
            (else
              (if (eqv? obj pat)
                  vals
                  (fail)))))))))
))

#!eof

;;; examples of passing along threaded information.

;;; Try (collect-symbols '(if (x y 'a 'c zz) 'b 'c))
;;; Note that it commonizes the reference to c.

(define-syntax with-values
  (syntax-rules ()
    ((_ P C) (call-with-values (lambda () P) C))))
(define collect-symbols
  (lambda (exp)
    (with-values (collect-symbols-help exp)
      (lambda (symbol-decls exp)
        (match symbol-decls
          (((,symbol-name . ,symbol-var) ...)
           `(let ((,symbol-var (quote ,symbol-name)) ...) ,exp)))))))
(define collect-symbols-help
  (lambda (exp)
    (let ((symbol-env '()))
      (match+ (symbol-env) exp
        (,x
          (guard (symbol? x))
          (values symbol-env x))
        ((quote ,x)
         (guard (symbol? x))
         (let ((pair/false (assq x symbol-env)))
           (if pair/false
               (values symbol-env (cdr pair/false))
               (let ((v (gensym)))
                 (values (cons (cons x v) symbol-env)
                         v)))))
        ((quote ,x)
         (values symbol-env `(quote ,x)))
        ((if ,[t] ,[c] ,[a])
         (values symbol-env `(if ,t ,c ,a)))
        ((,[op] ,[arg] ...)
         (values symbol-env `(,op ,arg ...)))))))

;;; the grammar for this one is just if-exprs and everything else

(define collect-leaves
  (lambda (exp acc)
    (match+ (acc) exp
      ((if ,[] ,[] ,[])
       acc)
      ((,[] ,[] ...)
       acc)
      (,x
        (cons x acc)))))

;; here's something that takes apart quoted stuff.
```

```
(define destruct
  (lambda (datum)
    (match datum
      (() `'())
      ((,[X] . ,[Y])`(cons ,X ,Y))
      (#(,[X] ...) `(vector ,X ...))
      (,thing
        (guard (symbol? thing))
        `',thing)
      (,thing
        thing))))

;; examples using explicit Catas

(define sumsquares
  (lambda (ls)
    (define square
      (lambda (x)
        (* x x)))
    (match ls
      [(,[a*] ...) (apply + a*)]
      [,[square -> n] n])))

(define sumsquares
  (lambda (ls)
    (define square
      (lambda (x)
        (* x x)))
    (let ([acc 0])
      (match+ (acc) ls
        [(,[] ...) acc]
        [,[(lambda (acc x) (+ acc (square x))) ->] acc]))))

;;; The following uses explicit Catas to parse programs in the
;;; simple language defined by the grammar below

;;; <Prog> -> (program <Stmt>* <Expr>)
;;; <Stmt> -> (if <Expr> <Stmt> <Stmt>)
;;;         | (set! <var> <Expr>)
;;; <Expr> -> <var>
;;;         | <integer>
;;;         | (if <Expr> <Expr> <Expr>)
;;;         | (<Expr> <Expr*>)

(define parse
  (lambda (x)
    (define Prog
      (lambda (x)
        (match x
          [(program ,[Stmt -> s*] ... ,[Expr -> e])
           `(begin ,s* ... ,e)]
          [,other (error 'parse "invalid␣program␣~s" other)])))
    (define Stmt
      (lambda (x)
        (match x
          [(if ,[Expr -> e] ,[Stmt -> s1] ,[Stmt -> s2])
           `(if ,e ,s1 ,s2)]
          [(set! ,v ,[Expr -> e])
           (guard (symbol? v))
```

```
           `(set! ,v ,e)]
          [,other (error 'parse "invalid␣statement␣~s" other)])))
    (define Expr
      (lambda (x)
        (match x
          [,v (guard (symbol? v)) v]
          [,n (guard (integer? n)) n]
          [(if ,[e1] ,[e2] ,[e3])
           `(if ,e1 ,e2 ,e3)]
          [(,[rator] ,[rand*] ...) `(,rator ,rand* ...)]
          [,other (error 'parse "invalid␣expression␣~s" other)])))
    (Prog x)))
;;; (parse '(program (set! x 3) (+ x 4)))) => (begin (set! x 3) (+ x 4))

;; CHANGELOG

;; [31 January 2010]
;; rkd replaced _ with k in the syntax-case patterns for match, match+,
;; etc., since in R6RS, _ is not a pattern variable.

;; [31 January 2010]
;; rkd renamed syntax-object->datum and datum->syntax-object to their
;; R6RS names syntax->datum and datum->syntax.  also replaced the
;; literal-identifier=? calls with free-identifier=? calls.

;; [3 February 2008]
;; rkd modified overloaded quasiquote to handle expressions followed
;; by more than one ellipsis.

;; [3 February 2008]
;; aziz modified mapper to quote the inserted empty lists

;; [3 March 2007]
;; aziz minor change to eagerly catch malformed clauses (e.g. a clause
;; that's not a list of 2 or more subforms).

;; [13 March 2002]
;; rkd added following change by Friedman and Ganz to the main source
;; code thread and fixed a couple of minor problems.

;; [9 March 2002]
;; Dan Friedman and Steve Ganz added the ability to use identical pattern
;; variables.  The patterns represented by the variables are compared
;; using the value of the parameter match-equality-test, which defaults
;; to equal?.
;;
;; > (match '(1 2 1 2 1)
;;     [(,a ,b ,a ,b ,a) (guard (number? a) (number? b)) (+ a b)])
;; 3
;; ;;
;; > (match '((1 2 3) 5 (1 2 3))
;;     [((,a ...) ,b (,a ...)) `(,a ... ,b)])
;; (1 2 3 5)
;; ;;
;; > (parameterize ([match-equality-test (lambda (x y) (equal? x (reverse y)))])
;;     (match '((1 2 3) (3 2 1))
;;       [(,a ,a) 'yes]
;;       [,oops 'no]))
;; yes
```

```
;; [10 Jan 2002]
;; eh fixed bug that caused (match '((1 2 3 4)) (((,a ... ,d) . ,x) a)) to
;; blow up.  The bug was caused by a bug in the sexp-dispatch procedure
;; where a base value empty list was passed to an accumulator from inside
;; the recursion, instead of passing the old value of the accumulator.

;; [14 Jan 2001]
;; rkd added syntax checks to unquote pattern parsing to weed out invalid
;; patterns like ,#(a) and ,[(vector-ref d 1)].

;; [14 Jan 2001]
;; rkd added ,[Cata -> Id* ...] to allow specification of recursion
;; function.  ,[Id* ...] recurs to match; ,[Cata -> Id* ...] recurs
;; to Cata.

;; [14 Jan 2001]
;; rkd tightened up checks for ellipses and nested quasiquote; was comparing
;; symbolic names, which, as had been noted in the source, is a possible
;; hygiene bug.  Replaced error call in guard-body with syntax-error to
;; allow error to include source line/character information.

;; [13 Jan 2001]
;; rkd fixed match patterns of the form (stuff* ,[x] ... stuff+), which
;; had been recurring on subforms of each item rather than on the items
;; themselves.

;; [29 Feb 2000]
;; Fixed a case sensitivity bug.

;; [24 Feb 2000]
;; Matcher now handles vector patterns.  Quasiquote also handles
;; vector patterns, but does NOT do the csv6.2 optimization of
;; `#(a 1 ,(+ 3 4) x y) ==> (vector 'a 1 (+ 3 4) 'x 'y).
;; Also fixed bug in (P ... . P) matching code.

;; [23 Feb 2000]
;; KSM fixed bug in unquote-splicing inside quasiquote.

;; [10 Feb 2000]
;; New forms match+ and trace-match+ thread arguments right-to-left.
;; The pattern (P ... . P) now works the way you might expect.
;; Infinite lists are now properly matched (and not matched).
;; Removed the @ pattern.
;; Internal: No longer converting into syntax-case.

;; [6 Feb 2000]
;; Added expansion-time error message for referring to cata variable
;; in a guard.

;; [4 Feb 2000]
;; Fixed backquote so it can handle nested backquote (oops).
;; Double-backquoted elipses are neutralized just as double-backquoted
;; unquotes are.  So:
;;   `(a ,'(1 2 3) ... b)    =eval=> (a 1 2 3 b)
;;   ``(a ,'(1 2 3) ... b)   =eval=> `(a ,'(1 2 3) ... b)
;;   ``(a ,(,(1 2 3) ...) b) =eval=> `(a ,(1 2 3) b)
;; Added support for
;;   `((unquote-splicing x y z) b) =expand==> (append x y z (list 'b))
```

```
905
;; [1 Feb 2000]
;; Fixed a bug involving forgetting to quote stuff in the revised backquote.
;; Recognized unquote-splicing and signalled errors in the appropriate places.
;; Added support for deep elipses in backquote.
910 ;; Rewrote backquote so it does the rebuilding directly instead of
;; expanding into Chez's backquote.

;; [31 Jan 2000]
;; Kent Dybvig fixed template bug.
915
;; [31 Jan 2000]
;; Added the trace-match form, and made guards contain
;; an explicit and expression:
;;    (guard E ...) ==> (guard (and E ...))
920
;; [26 Jan 2000]
;; Inside the clauses of match expressions, the following
;; transformation is performed inside backquote expressions:
;;    ,v ...       ==> ,@v
925 ;;    (,v ,w) ... ==> ,@(map list v w)
;;    etc.
```

Bibliography

Michael D. Adams, Andrew W. Keep, Jan Midtgaard, Matthew Might, Arun Chauhan, and R. Kent Dybvig. Flow-sensitive type recovery in linear-log time. In *Proceedings of the ACM international conference on Object oriented programming systems languages and applications*, OOPSLA '11, New York, NY, USA, October 2011. ACM. To appear.

Norman Adams, David Kranz, Richard Kelsey, Jonathan Rees, Paul Hudak, and James Philbin. ORBIT: An optimizing compiler for Scheme. In *Proceedings of the 1986 SIGPLAN symposium on Compiler construction*, SIGPLAN '86, pages 219–233, New York, NY, USA, 1986. ACM. ISBN 0-89791-197-0. doi: 10.1145/12276.13333.

Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. Time and tape complexity of pushdown automaton languages. *Information and Control*, 13(3):186–206, September 1968. ISSN 0019-9958. doi: 10.1016/S0019-9958(68)91087-5.

Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. On finding lowest common ancestors in trees. In *Proceedings of the fifth annual ACM symposium on Theory of computing*, STOC '73, pages 253–265, New York, NY, USA, 1973. ACM. doi: 10.1145/800125.804056.

Bowen Alpern, Mark N. Wegman, and F. Kenneth Zadeck. Detecting equality of variables in programs. In *Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Prin-*

ciples of programming languages, POPL '88, pages 1–11, New York, NY, USA, 1988. ACM. ISBN 0-89791-252-7. doi: 10.1145/73560.73561.

Stephen Alstrup, Cyril Gavoille, Haim Kaplan, and Theis Rauhe. Nearest common ancestors: A survey and a new algorithm for a distributed environment. *Theory of Computing Systems*, 37(3):441–456, May 2004. ISSN 1432-4350. doi: 10.1007/s00224-004-1155-5.

Andrew W. Appel and Trevor Jim. Continuation-passing, closure-passing style. In *Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '89, pages 293–302, New York, NY, USA, 1989. ACM. ISBN 0-89791-294-2. doi: 10.1145/75277.75303.

J. Michael Ashley and R. Kent Dybvig. A practical and flexible flow analysis for higher-order languages. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 20(4):845–868, July 1998. ISSN 0164-0925. doi: 10.1145/291891.291898.

John P. Banning. An efficient way to find the side effects of procedure calls and the aliases of variables. In *Proceedings of the 6th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, POPL '79, pages 29–41, New York, NY, USA, 1979. ACM. doi: 10.1145/567752.567756.

Robert Cartwright and Mike Fagan. Soft typing. In *Proceedings of the ACM SIGPLAN 1991 conference on Programming language design and implementation*, PLDI '91, pages 278–292, New York, NY, USA, 1991. ACM. ISBN 0-89791-428-7. doi: 10.1145/113445.113469.

Swarat Chaudhuri. Subcubic algorithms for recursive state machines. In *Proceedings of the 35th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '08, pages 159–169, New York, NY, USA, 2008. ACM. ISBN 978-1-59593-689-9. doi: 10.1145/1328438.1328460.

William D. Clinger. Description of benchmarks, 2008. URL `http://www.larcenists.org/benchmarksAboutR6.html`.

Patrick Cousot. Semantic foundations of program analysis. In Steven S. Muchnick and Neil D. Jones, editors, *Program Flow Analysis: Theory and Applications*, chapter 10, pages 303–342. Prentice-Hall, 1981.

Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, POPL '77, pages 238–252, New York, NY, USA, 1977. ACM. doi: 10.1145/512950.512973.

Patrick Cousot and Radhia Cousot. Systematic design of program analysis frameworks. In *Proceedings of the 6th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, POPL '79, pages 269–282, New York, NY, USA, 1979. ACM. doi: 10.1145/567752.567778.

Patrick Cousot and Radhia Cousot. Abstract interpretation frameworks. *Journal of Logic and Computation*, 2(4):511–547, August 1992. ISSN 0955-792X/1465-363X. doi: 10.1093/logcom/2.4.511.

Patrick Cousot and Radhia Cousot. Refining model checking by abstract interpretation. *Automated Software Engineering*, 6(1):69–95, January 1999. ISSN 0928-8910. doi: 10.1023/A:1008649901864.

Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 13(4):451–490, October 1991. ISSN 0164-0925. doi: 10.1145/115372.115320.

Manuvir Das, Sorin Lerner, and Mark Seigle. ESP: Path-sensitive program verification in polynomial time. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, PLDI '02, pages 57–68, New York, NY, USA, 2002. ACM. ISBN 1-58113-463-0. doi: 10.1145/512529.512538.

R. Kent Dybvig. *Chez Scheme Version 8 User's Guide*. Cadence Research Systems, 2010.

R. Kent Dybvig, Robert Hieb, and Carl Bruggeman. Syntactic abstraction in Scheme. *LISP and Symbolic Computation*, 5(4):295–326, December 1993. ISSN 0892-4635. doi: 10.1007/BF01806308.

Marc Feeley. SRFI 39: Parameter objects, June 2003. URL `http://srfi.schemers.org/srfi-39/srfi-39.html`.

Emden R. Gansner and Stephen C. North. An open graph visualization system and its applications to software engineering. *Software: Practice and Experience*, 30(11): 1203–1233, 2000. ISSN 1097-024X. doi: 10.1002/1097-024X(200009)30:11<1203::AID-SPE338>3.0.CO;2-N.

Arjun Guha, Claudiu Saftoiu, and Shriram Krishnamurthi. Typing local control and state using flow analysis. In Gilles Barthe, editor, *Programming Languages and Systems*, volume 6602 of *Lecture Notes in Computer Science*, pages 256–275. Springer Berlin / Heidelberg, 2011. ISBN 978-3-642-19717-8. doi: 10.1007/978-3-642-19718-5_14.

Nevin Heintze and David McAllester. On the cubic bottleneck in subtyping and flow analysis. *Logic in Computer Science, Symposium on*, page 342, 1997a. ISSN 1043-6871. doi: 10.1109/LICS.1997.614960.

Nevin Heintze and David McAllester. On the complexity of set-based analysis. In *Proceedings of the second ACM SIGPLAN international conference on Functional programming*,

ICFP '97, pages 150–163, New York, NY, USA, 1997b. ACM. ISBN 0-89791-918-1. doi: 10.1145/258948.258963.

Nevin Heintze and David McAllester. Linear-time subtransitive control flow analysis. In *Proceedings of the ACM SIGPLAN 1997 conference on Programming language design and implementation*, PLDI '97, pages 261–272, New York, NY, USA, 1997c. ACM. ISBN 0-89791-907-6. doi: 10.1145/258915.258939.

Fritz Henglein. Global tagging optimization by type inference. *ACM SIGPLAN Lisp Pointers*, V(1):205–215, January 1992a. ISSN 1045-3563. doi: 10.1145/141478.141542.

Fritz Henglein. Simple closure analysis. Semantics Report D-193, DIKU, University of Copenhagen, 1992b.

Fritz Henglein. Dynamic typing: Syntax and proof theory. *Science of Computer Programming*, 22(3):197–230, June 1994. ISSN 0167-6423. doi: 10.1016/0167-6423(94)00004-2.

Suresh Jagannathan and Stephen Weeks. A unified treatment of flow analysis in higher-order languages. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '95, pages 393–407, New York, NY, USA, 1995. ACM. ISBN 0-89791-692-1. doi: 10.1145/199448.199536.

Suresh Jagannathan, Peter Thiemann, Stephen Weeks, and Andrew Wright. Single and loving it: Must-alias analysis for higher-order languages. In *Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '98, pages 329–341, New York, NY, USA, 1998. ACM. ISBN 0-89791-979-3. doi: 10.1145/268946.268973.

Simon Jensen, Anders Møller, and Peter Thiemann. Type analysis for JavaScript. In Jens Palsberg and Zhendong Su, editors, *Static Analysis*, volume 5673 of *Lecture Notes in*

Computer Science, pages 238–255. Springer Berlin / Heidelberg, 2009. ISBN 978-3-642-03236-3. doi: 10.1007/978-3-642-03237-0_17.

Donald E. Knuth. Semantics of context-free languages. *Theory of Computing Systems*, 2 (2):127–145, June 1968. ISSN 1432-4350. doi: 10.1007/BF01692511.

Ondrej Lhoták and Kwok-Chiang Andrew Chung. Points-to analysis with efficient strong updates. In *Proceedings of the 38th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '11, pages 3–16, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0490-0. doi: 10.1145/1926385.1926389.

Thomas J. Marlowe, Barbara Ryder, and Michael Burke. Defining flow sensitivity in data flow problems. Research Report RC 20138, IBM T. J. Watson Research Center, July 1995.

David Melski and Thomas Reps. Interconvertibility of a class of set constraints and context-free-language reachability. *Theoretical Computer Science*, 248(1–2):29–98, October 2000. ISSN 0304-3975. doi: 10.1016/S0304-3975(00)00049-9.

Jan Midtgaard and Thomas Jensen. A calculational approach to control-flow analysis by abstract interpretation. In María Alpuente and Germán Vidal, editors, *Static Analysis*, volume 5079 of *Lecture Notes in Computer Science*, pages 347–362. Springer Berlin / Heidelberg, 2008. ISBN 978-3-540-69163-1. doi: 10.1007/978-3-540-69166-2_23.

Jan Midtgaard and David Van Horn. Subcubic control flow analysis algorithms. Computer Science Research Report 125, Roskilde University, Roskilde, Denmark, May 2009. Revised version to appear in Higher-Order and Symbolic Computation.

Matthew Might and Olin Shivers. Improving flow analyses via ΓCFA: Abstract garbage collection and counting. In *Proceedings of the eleventh ACM SIGPLAN international*

conference on Functional programming, ICFP '06, pages 13–25, New York, NY, USA, 2006. ACM. ISBN 1-59593-309-3. doi: 10.1145/1159803.1159807.

Torben Æ. Mogensen. Glossary for partial evaluation and related topics. *Higher-Order and Symbolic Computation*, 13(4):355–368, December 2000. ISSN 1388-3690. doi: 10.1023/A:1026551132647.

Christian Mossin. Higher-order value flow graphs. *Nordic Journal of Computing*, 5(3):214–234, September 1998. ISSN 1236-6064. URL `http://dl.acm.org/citation.cfm?id=640057.640060`.

Mozilla Corporation. Doctor JS, 2011. `http://doctorjs.org/`.

Eugene W. Myers. Efficient applicative data types. In *Proceedings of the 11th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, POPL '84, pages 66–75, New York, NY, USA, 1984. ACM. ISBN 0-89791-125-3. doi: 10.1145/800017.800517.

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

Simon Peyton Jones, Will Partain, and André Santos. Let-floating: Moving bindings to give faster programs. In *Proceedings of the first ACM SIGPLAN international conference on Functional programming*, ICFP '96, pages 1–12, New York, NY, USA, 1996. ACM. ISBN 0-89791-770-7. doi: 10.1145/232627.232630.

William Pugh. Skip lists: A probabilistic alternative to balanced trees. *Communications of the ACM*, 33(6):668–676, June 1990. ISSN 0001-0782. doi: 10.1145/78973.78977.

Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Global value numbers and redundant computations. In *Proceedings of the 15th ACM SIGPLAN-SIGACT sympo-*

sium on Principles of programming languages, POPL '88, pages 12–27, New York, NY, USA, 1988. ACM. ISBN 0-89791-252-7. doi: 10.1145/73560.73562.

Manuel Serrano. Control flow analysis: A functional languages compilation paradigm. In *Proceedings of the 1995 ACM symposium on Applied computing*, SAC '95, pages 118–122, New York, NY, USA, 1995. ACM. ISBN 0-89791-658-1. doi: 10.1145/315891.315934.

Olin Shivers. Control flow analysis in Scheme. In *Proceedings of the ACM SIGPLAN 1988 conference on Programming Language design and Implementation*, PLDI '88, pages 164–174, New York, NY, USA, 1988. ACM. ISBN 0-89791-269-1. doi: 10.1145/53990.54007.

Olin Shivers. Data-flow analysis and type recovery in Scheme. In Peter Lee, editor, *Topics in Advanced Language Implementations*, pages 47–87. The MIT Press, 1991.

Jeremy G. Siek and Walid Taha. Gradual typing for functional languages. In *Scheme and Functional Programming Workshop*, pages 81–92, September 2006.

Michael Sperber, R. Kent Dybvig, Matthew Flatt, and Anton van Straaten (eds.). Revised^6 report on the algorithmic language Scheme, September 2007. URL `http://www.r6rs.org/`.

Peter A. Steenkiste. The implementation of tags and run-time type checking. In Peter Lee, editor, *Topics in Advanced Language Implementations*, pages 3–24. The MIT Press, 1991.

Sam Tobin-Hochstadt and Matthias Felleisen. Logical types for untyped languages. In *Proceedings of the 15th ACM SIGPLAN international conference on Functional programming*, ICFP '10, pages 117–128, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-794-3. doi: 10.1145/1863543.1863561.

David Van Horn and Harry G. Mairson. Deciding *k*CFA is complete for EXPTIME. In *Proceeding of the 13th ACM SIGPLAN international conference on Functional programming*, ICFP '08, pages 275–282, New York, NY, USA, 2008. ACM. ISBN 978-1-59593-919-7. doi: 10.1145/1411204.1411243.

Dimitrios Vardoulakis and Olin Shivers. CFA2: A context-free approach to control-flow analysis. In Andrew Gordon, editor, *Programming Languages and Systems*, volume 6012 of *Lecture Notes in Computer Science*, pages 570–589. Springer Berlin / Heidelberg, 2010. ISBN 978-3-642-11956-9. doi: 10.1007/978-3-642-11957-6_30.

Oscar Waddell and R. Kent Dybvig. Fast and effective procedure inlining. In Pascal Van Hentenryck, editor, *Static Analysis*, volume 1302 of *Lecture Notes in Computer Science*, pages 35–52. Springer Berlin / Heidelberg, 1997. ISBN 978-3-540-63468-3. doi: 10.1007/BFb0032732.

Oscar Waddell, Dipanwita Sarkar, and R. Dybvig. Fixing letrec: A faithful yet efficient implementation of Scheme's recursive binding construct. *Higher-Order and Symbolic Computation*, 18(3):299–326, December 2005. ISSN 1388-3690. doi: 10.1007/s10990-005-4878-3.

Mark N. Wegman and F. Kenneth Zadeck. Constant propagation with conditional branches. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 13 (2):181–210, April 1991. ISSN 0164-0925. doi: 10.1145/103135.103136.

Index of Source Files

ast-bubbles.ss, 134

ast-flow-graphs.ss, 176

ast.ss, 167

ast-unparse.ss, 162

bitypes.ss, 185

composition-stacks.ss, 141

datatype.ss, 197

define-ignored.ss, 201

driver.ss, 124

env-flow-functions.ss, 186

env-flow-graphs.ss, 144

flow-graphs.ss, 171

idents.ss, 166

iota.ss, 201

match.ss, 205

myers-stacks.ss, 138

parse-scheme.ss, 124

prims.ss, 189

records-io.ss, 202

safe-forms.ss, 202

srfi-39.ss, 204

syntax-helpers.ss, 203

tests.ss, 130

tprogram.ss, 167

transform-let.ss, 133

tree-locations.ss, 136

type-recovery.ss, 124

types.ss, 179

value-flow-graphs.ss, 146

Index of Functions

`$check-index`, myers-stacks.ss:129

`$check-length`, myers-stacks.ss:131

`$check-range`, myers-stacks.ss:126

`$iota`, iota.ss:8

`add-node!`, value-flow-graphs.ss:371

`add-rest`, prims.ss:81

`add-to-join-listeners!`,
flow-graphs.ss:303

`add-to-listeners!`, flow-graphs.ss:299

`add-to-pending!`, flow-graphs.ss:294

`alpha-equiv-clause`, tests.ss:10

`alpha-equiv-var?`, tests.ss:7

`alpha-equiv?`, tests.ss:6

`app-out-reachable-by-env`,
value-flow-graphs.ss:893

`app-out-reachable-by-value`,
value-flow-graphs.ss:883

`arity-match?`, ast-unparse.ss:9

`AST->bare-var-node`,
value-flow-graphs.ss:200

`AST->bool-node`, value-flow-graphs.ss:86

`AST->Trex`, ast-unparse.ss:151

`AST->type-id-node`,
value-flow-graphs.ss:59

`AST-bubble-vars-add!`,
ast-bubbles.ss:48

`AST-bubble-vars-contains?`,
ast-bubbles.ss:51

`AST-bubbles-intersect`,
ast-bubbles.ss:8

`AST-bypass-nodes`,
value-flow-graphs.ss:486

`AST-contained-nodes-add!`,
ast-flow-graphs.ss:46

`AST-flow-graph`, ast-flow-graphs.ss:30

`AST-flow-graph-print`,
ast-flow-graphs.ss:54
`AST-output-type`, ast-unparse.ss:149
`AST-Ref->ident`,
value-flow-graphs.ss:194

`bitype->type`, bitypes.ss:37
`bitype->void-bitype`, bitypes.ss:47
`bitype-false->void-bitype`,
bitypes.ss:50
`bitype-intersect`, bitypes.ss:56
`bitype-true->void-bitype`,
bitypes.ss:49
`bitype-union`, bitypes.ss:52
`bitype=?`, bitypes.ss:33
`bool-node-enable!`,
value-flow-graphs.ss:64
`bot-type?`, types.ss:240
`bubble-base`, ast-bubbles.ss:55

`calculate-clause`,
value-flow-graphs.ss:658
`case-lambda-clause-ref`, ast.ss:113
`CaseLambdaClause->AST`, ast.ss:123
`check-prim`, ast-unparse.ss:119
`check-sig`, ast-unparse.ss:116
`composition-stack-cons`,
composition-stacks.ss:96
`composition-stack-jump`,
composition-stacks.ss:82
`composition-stack-range`,
composition-stacks.ss:117
`constant?`, parse-scheme.ss:125

`datatype-property-set!`,
datatype.ss:170
`datum?`, parse-scheme.ss:128

`ellipsis?`, ast-unparse.ss:39
`env-flow-function-apply`,
env-flow-functions.ss:98
`env-flow-function-compose`,
env-flow-functions.ss:79
`env-flow-function=?`,
env-flow-functions.ss:75
`env-flow-graph-force!`,
env-flow-graphs.ss:99
`env-flow-source-apply`,
env-flow-functions.ss:99
`escape-label`, flow-graphs.ss:217
`escape-proc`, value-flow-graphs.ss:155
`escape-top-level-lambda`,
type-recovery.ss:7
`Expr`, parse-scheme.ss:160

`extend`, parse-scheme.ss:142

`extend-variables`, parse-scheme.ss:145

`false-type?`, types.ss:247

`find-arity`, ast-unparse.ss:41

`flow-graph-lazy-node-action`,
flow-graphs.ss:94

`flow-graph-mutable-node-action`,
flow-graphs.ss:138

`flow-graph-mutable-node-set!`,
flow-graphs.ss:146

`flow-graph-mutable-node?`,
flow-graphs.ss:142

`flow-graph-pending-add!`,
flow-graphs.ss:53

`flow-graph-pending-remove!`,
flow-graphs.ss:57

`flow-graph-print`, flow-graphs.ss:193

`flow-graph-stable?`, flow-graphs.ss:68

`flow-graph-step!`, flow-graphs.ss:62

`ident->string`, ast-flow-graphs.ss:58

`ident-preceding-ref-delete!`,
idents.ss:34

`ident<?`, idents.ss:32

`ident=?`, idents.ss:31

`ident>?`, idents.ss:33

`improper-list-of`, datatype.ss:26

`improper-list-split`, ast.ss:124

`improper-map`, parse-scheme.ss:19

`improper-memq`, parse-scheme.ss:33

`indent`, flow-graphs.ss:196

`inflate`, ast-bubbles.ss:78

`input-argument-proc`,
value-flow-graphs.ss:169

`input-called-proc`,
value-flow-graphs.ss:158

`intersected-node`,
value-flow-graphs.ss:448

`iota`, iota.ss:6

`key-locations<=?`, tree-locations.ss:64

`labeler`, ast-flow-graphs.ss:129

`list->composition-stack`,
composition-stacks.ss:113

`list->myers-stack`, myers-stacks.ss:108

`list-of`, datatype.ss:20

`lookup`, parse-scheme.ss:143

`make-AST-binding-node`,
value-flow-graphs.ss:66

`make-AST-binding-node-add!`,
value-flow-graphs.ss:81

`make-AST-bitype-id-node`, value-flow-graphs.ss:56
`make-AST-bubbles`, ast-bubbles.ss:17
`make-AST-const-node`, prims.ss:6
`make-AST-const-node`, value-flow-graphs.ss:47
`make-AST-const-type-guard-node`, value-flow-graphs.ss:52
`make-AST-enable-node`, value-flow-graphs.ss:62
`make-AST-type-guard-node`, value-flow-graphs.ss:49
`make-AST-var-binding-node`, value-flow-graphs.ss:69
`make-case-lambda-table`, ast.ss:87
`make-clause`, datatype.ss:52
`make-composition-stack-null`, composition-stacks.ss:91
`make-const-proc`, value-flow-graphs.ss:501
`make-env-flow-graph`, env-flow-graphs.ss:13
`make-flow-graph-lazy-node`, flow-graphs.ss:96
`make-flow-graph-mutable-node`, flow-graphs.ss:140
`make-out-formal`, value-flow-graphs.ss:642
`make-out-return`, value-flow-graphs.ss:117
`make-parameter`, srfi-39.ss:5
`make-proc-type`, types.ss:227
`make-seq`, parse-scheme.ss:88
`make-seq`, parse-scheme.ss:94
`make-uvar-ident`, parse-scheme.ss:136
`make-value-flow-graph`, value-flow-graphs.ss:492
`map-values`, ast-unparse.ss:131
`mark-escaped-lambda!`, types.ss:295
`mark-escaped-lambda!`, types.ss:305
`mark-escaped-type!`, types.ss:328
`mark-safe?`, ast-unparse.ss:7
`maybe-non-proc-type->type`, types.ss:347
`move-var-to-here`, value-flow-graphs.ss:466
`myers-stack->list`, myers-stacks.ss:111
`myers-stack-car`, myers-stacks.ss:74
`myers-stack-car-set!`, myers-stacks.ss:72
`myers-stack-cdr`, myers-stacks.ss:78
`myers-stack-cons`, myers-stacks.ss:93

`myers-stack-drop`, myers-stacks.ss:172
`myers-stack-eq-suffix`,
myers-stacks.ss:187
`myers-stack-find`, myers-stacks.ss:137
`myers-stack-null?`, myers-stacks.ss:90
`myers-stack-pair?`, myers-stacks.ss:105
`myers-stack-previous`,
myers-stacks.ss:180
`myers-stack-previous-ref`,
myers-stacks.ss:183
`myers-stack-ref`, myers-stacks.ss:118
`myers-stack-suffix`,
myers-stacks.ss:167

`name`, env-flow-graphs.ss:22
`name`, value-flow-graphs.ss:141
`node-cluster`, flow-graphs.ss:207
`node-id`, flow-graphs.ss:204
`node-name`, flow-graphs.ss:210
`node-output-set!`, flow-graphs.ss:273
`node-table-add-node!`,
flow-graphs.ss:201
`non-bot-type?`, types.ss:241
`normalize-field-spec`, datatype.ss:98
`normalize-property-spec`,
datatype.ss:92
`not-type`, types.ss:344
`null-type?`, types.ss:252

`obj->string`, ast-flow-graphs.ss:55
`output-argument-proc`,
value-flow-graphs.ss:187
`output-return-proc`,
value-flow-graphs.ss:182

`pair-type?`, types.ss:250
`parallel`, value-flow-graphs.ss:363
`parallel-one`, value-flow-graphs.ss:364
`parse-scheme`, parse-scheme.ss:269
`parse-unsafe-scheme`,
parse-scheme.ss:270
`prim-name->type`, prims.ss:493
`print-containment`, flow-graphs.ss:220
`print-node`, flow-graphs.ss:230
`printer`, ast-flow-graphs.ss:64
`proc-record->type`, types.ss:235
`proc-type-bits-intersect`,
types.ss:270
`proc-type-bits-union`, types.ss:282
`proc-type-bits=?`, types.ss:204
`proc-type-bits?`, types.ss:179

`rec-bypass`, env-flow-graphs.ss:41
`rec-id`, env-flow-graphs.ss:37

`rec-if-branch`, env-flow-graphs.ss:53

`rec-if-test`, env-flow-graphs.ss:49

`rec-true`, value-flow-graphs.ss:544

`rec-void`, env-flow-graphs.ss:45

`Ref->ident`, parse-scheme.ss:273

`remove-from-join-listeners!`,
flow-graphs.ss:308

`rest-return-type`,
value-flow-graphs.ss:118

`return-values`, ast-unparse.ss:132

`run`, driver.ss:5

`run-tests`, tests.ss:174

`run-type-recovery`, type-recovery.ss:11

`source`, ast.ss:118

`split-formals&rest`,
value-flow-graphs.ss:102

`strip-begin-seq`, parse-scheme.ss:89

`syntax-append`, syntax-helpers.ss:46

`syntax-car`, syntax-helpers.ss:29

`syntax-cdr`, syntax-helpers.ss:30

`syntax-ellipsis?`, syntax-helpers.ss:11

`syntax-for-all`, syntax-helpers.ss:39

`syntax-indexes`, syntax-helpers.ss:14

`syntax-length`, syntax-helpers.ss:24

`syntax-map`, syntax-helpers.ss:32

`transform-let`, transform-let.ss:6

`tree-location-table-next-ref`,
tree-locations.ss:87

`tree-location-table-prev-ref`,
tree-locations.ss:80

`Trex->AST`, ast.ss:117

`true-type?`, types.ss:245

`turn-var-down!`,
value-flow-graphs.ss:221

`turn-var-up!`, value-flow-graphs.ss:212

`type->bitype`, bitypes.ss:40

`type->bitype-true`, bitypes.ss:41

`type->proc-record*`,
value-flow-graphs.ss:515

`type->split-bitype`, bitypes.ss:43

`type-filter`, types.ss:337

`type-intersect`, types.ss:254

`type-match?`, ast-unparse.ss:13

`type-node-add!`, value-flow-graphs.ss:77

`type-of`, types.ss:350

`type-union`, types.ss:292

`type-with-rest-match?`,
ast-unparse.ss:16

`type<=?`, types.ss:94

`type=?`, types.ss:90

`types-match?`, ast-unparse.ss:26

`unique-name`, parse-scheme.ss:10

`unique-var`, parse-scheme.ss:139

`unparse-scheme`, parse-scheme.ss:272

`uvar-maker`, parse-scheme.ss:135

`vector-binary-search`,
tree-locations.ss:67

`vector-of`, datatype.ss:23

`vector-set-or!`, ast.ss:88

`verify-var-list`, parse-scheme.ss:111

`wrap-var`, parse-scheme.ss:138

Index of Macros

`$make-env-flow-source`, env-flow-functions.ss:54

`bot-proc-type-bits`, types.ss:161

`build-prim-entry`, ast-unparse.ss:31

`case`, safe-forms.ss:44

`case-test`, safe-forms.ss:66

`clause-body`, match.ss:173

`cond`, safe-forms.ss:17

`cond-env-flow-function`, env-flow-functions.ss:62

`convert-pat`, match.ss:216

`define-case`, datatype.ss:32

`define-datatype`, datatype.ss:85

`define-ignored`, define-ignored.ss:5

`define-mask-bits`, types.ss:106

`define-prim-table`, ast-unparse.ss:37

`define-proc-fun`, value-flow-graphs.ss:138

`define-rec`, env-flow-graphs.ss:17

`extend-backquote`, match.ss:528

`flatten-mask-bits`, types.ss:98

`flow-graph-lazy-node-force!`, flow-graphs.ss:99

`flow-mask`, env-flow-functions.ss:39

`flow-masks`, env-flow-functions.ss:48

`fold-left*`, value-flow-graphs.ss:262

`for-each*`, value-flow-graphs.ss:278

`guard-body`, match.ss:201

`if`, safe-forms.ss:11

`let-synvalues*`, match.ss:320

`let-values**`, match.ss:121

`letcc`, match.ss:556

`list-rev`, value-flow-graphs.ss:245

`make-AST-node`, ast-flow-graphs.ss:37
`make-flow-graph-function-node`, flow-graphs.ss:120
`make-flow-graph-join-node`, flow-graphs.ss:153
`make-pred-prim`, prims.ss:107
`make-prim`, prims.ss:96
`make-prim-core`, prims.ss:9
`make-prim-rest`, prims.ss:79
`make-top-prim`, prims.ss:100
`make-tree-location-table`, tree-locations.ss:25
`map*`, value-flow-graphs.ss:300
`mapper`, match.ss:334
`match`, match.ss:96
`match+`, match.ss:89
`match-help`, match.ss:129
`match-help1`, match.ss:143
`my-backquote`, match.ss:356
`myers-stack-find-macro`, myers-stacks.ss:150

`one-or-more-proc-type-bits`, types.ss:174
`parallel*`, value-flow-graphs.ss:314
`parallel-intersected`, value-flow-graphs.ss:414
`parallel-list`, value-flow-graphs.ss:250
`parallel-loop`, value-flow-graphs.ss:231
`parameterize`, srfi-39.ss:19
`proc-type-bits-case`, types.ss:184

`select-CFA`, types.ss:73
`syntax-lambda`, match.ss:364

`top-proc-type-bits`, types.ss:168
`trace-match`, match.ss:111
`trace-match+`, match.ss:103

`unless-unchecked`, datatype.ss:12

`with-AST-flow-graph`, ast-flow-graphs.ss:32
`with-ellipsis-aware-quasiquote`, match.ss:539
`with-temp`, match.ss:372
`with-temps`, match.ss:377
`with-values`, match.ss:361
`with-values`, match.ss:552
`with-values`, match.ss:681

Index of Record Types

`bitype`, bitypes.ss:23

`case-lambda-clause`, ast.ss:72

`clause-info`, value-flow-graphs.ss:90

`composition-stack`,
composition-stacks.ss:77

`down-ref`, value-flow-graphs.ss:136

`flow-graph`, flow-graphs.ss:48

`flow-graph-node`, flow-graphs.ss:73

`ident`, idents.ss:16

`letrec-binding`, ast.ss:62

`myers-stack`, myers-stacks.ss:68

`proc-record`, types.ss:217

`tree-location-table`,
tree-locations.ss:9

`Trex`, tprogram.ss:10

`Trex-CaseLambdaClause`, tprogram.ss:7

`type`, types.ss:81

`up-ref`, value-flow-graphs.ss:135

Michael D. Adams

Present Address:
2001 E Lingelbach Ln./Apt. 325
Bloomington, IN 47408
+1-785-969-2431
adamsmd@indiana.edu
`http://www.cs.indiana.edu/~adamsmd/`

Permanent Address:
3626 SE Tomahawk Ct.
Topeka, KS 66605
+1-785-266-2778

Research Interests

Programming Languages, Compilation and Optimization, Static Analysis, Control-Flow Analysis, Type Systems, Dependent Types, Mechanized Logic, Program Verification

Education

Indiana University **Bloomington, Indiana**

Doctor of Philosophy in Computer Science with a Minor in Logic *October 2011*
Advisor: R. Kent Dybvig

The University of Kansas **Lawrence, Kansas**

Bachelor of Science in Computer Science and Bachelor of Science in Computer Engineering with a Minor in Mathematics *May 2005*

Professional Experience

Cadence Research (Dr. R. Kent Dybvig) **Bloomington, Indiana**

Independent Contractor *May 2008 – August 2010*

- Created a type recovery optimization for Chez Scheme based on control-flow analysis
- Helped update Chez Scheme to R6RS language standard
- Overhauled Chez I/O subsystem

Microsoft Research **Cambridge, England**

Intern *April 2007 – June 2007*

- Worked on Glasgow Haskell Compiler internals
- Added function call and procedure support to C-- language

IBM Research **Hawthorne, New York**

Intern *January 2007 – March 2007*

- Worked on X10 language
- Implemented ideas from Arcee in X10

Arcee Project (Dr. David S. Wise) **Bloomington, Indiana**

Graduate Research Assistant *August 2005 – December 2008*

- Designed techniques for faster linear algebra which forms the core of most simulations
- Doubled the speed of the research solver to be faster than other best of breed solvers
- Optimized the parallel-computing communication

JParse (Dr. Jerry James) **Lawrence, Kansas**

Software Engineer/Programmer *June 2004 – April 2005*

- Maintained a Java parser for source transforms
- Refactored the system to be multiphase to allow transforms to be more modular

Honors Undergraduate Research (Dr. Perry Alexandar) **Lawrence, Kansas**

Undergraduate Research Assistant *August 2003 – May 2004*

- Built a new embedded language for describing and checking design constraints

Veatros (Dr. John Gauch) **Lawrence, Kansas**

Software Engineer/Programmer *May 2002 – May 2004*

- Developed software for television commercial detection on remote machines

Die Kueche Café **Paxico, Kansas**

Independent Contractor *June 2000 – July 2000*

- Overhauled company web site

Western Resources **Topeka, Kansas**

Independent Contractor *July 1998 – June 1999*

- Designed and implemented a web-based weather-image display and cataloging site

Teaching Experience

Indiana University **Bloomington, Indiana**

Associate Instructor January 2009 – May 2011

- H212: Introduction to Software Systems, Honors: Spring 2011
- H211: Introduction to Computer Science, Honors: Fall 2010
- C343/A594: Data Structures: Fall 2009 and Spring 2010
- CSCI C212/A592: Introduction to Software Systems: Spring 2009

Undergraduate Research Opportunities in Computing January 2011 – May 2011

- Research Mentor for Christopher Zakian and Yitian Peng: Spring 2011

Honors and Awards

- Associate Instructor of the Year – Comp. Sci., Indiana University 2009 – 2010
- Paul and Virginia B. Miller Scholar – EECS, University of Kansas 2004 – 2005
- School of Engineering Honor Roll – University of Kansas Fall 2000 – Spring 2005
- Tau Beta Pi National Scholar Fall 2004
- Senior Everitt Award – EECS, University of Kansas Spring 2004
- W. Harold Otto National Merit Scholar 2000 – 2004
- May Landis Scholar – Mathematics, University of Kansas 2001 – 2002
- University Scholar Finalist (top 40 sophomores) 2001
- First Place in Archery at Sunflower State Games 1998

Recent Activities

Scheme Workshop 2011 – Program Committee **2011**

`http://scheme2011.ucombinator.org/`

PL Wonks **March 2007 – Present**

`http://lambda.soic.indiana.edu/`

- Member of and frequent speaker at weekly, programming-languages seminar

Crystal Space 3D **February 2005 – October 2006**

`http://www.crystalspace3d.org`

- A core developer of this Open-Source, 3D SDK
- Coauthor of the Shared Class Facility providing reference counting and interface querying
- Invited participant in the Crystal Space Conference 2006 in Aachen, Germany

Lambda Group **June 2003 – May 2005**

`http://wiki.ittc.ku.edu/lambda`

- Member of this readings group sponsored by Dr. Alexander
- Presented a session on fixed point combinators

Engineering Expo Chair for ACM **August 2004 – April 2005**

- Oversaw the development of multiple Expo projects to represent the Electrical Engineering and Computer Science Department

Tau Beta Pi – The Engineering Honor Society **November 2002 – May 2005**

- Recording secretary for 2004-2005 school year
- Delegate to the annual Regional Conference in Spring 2003 and Spring 2004.
- Delegate to the 2004 National Convention and member of the committee charged with drafting an amendment to the Tau Beta Pi constitution addressing Computer Science eligibility.

Honors Undergraduate Research (Dr. Perry Alexander) **June 2003 – May 2004**

- Developed a system for describing constraints over Abstract Syntax Trees
- Implemented first-class patterns in Haskell

Engineering Expo Project for IEEE **November 2003 – February 2004**

- Coordinated a project to display a game of Tic-Tac-Toe on the side of a building
- Board displayed on building using lights
- Lights are controlled by the X10 home automation technology
- Users play the game though a web based interface

Talks

- **Linear-Log Time Control-Flow Analysis with Flow-Sensitivity and Predicate-Awareness**. POPL Student Presentation Session, January 27, 2011.
- **Control Flow Analysis**. Algorithms Reading Group, Indiana University, September 20, 2010.
- **Scrap Your Zippers: A Generic Zipper for Heterogeneous Types**. Programming Languages Seminar, Indiana University, September 17, 2010.
- **Control-Flow Analysis and Abstract Interpretation: Running Your Program without Running Your Program**. Programming Languages Seminar, Indiana University, October 2, 2009.
- **Deriving Syntax-Case from First Principals: How to turn a name freshener into a macro expander**. Programming Languages Seminar, Indiana University, February 6, 2009.
- **Verifying a Filesystem: A Successful Failure**. (Co-speaker with Joseph Near and Aaron Kahn.) Programming Languages Seminar, Indiana University, April 25, 2008.
- **Scrapping Scrap your Nameplate**. Programming Languages Seminar, Indiana University, March 7, 2008.
- **Meta-Programming in Haskell with GADTs**. Programming Languages Seminar, Indiana University, October 26, 2007.
- **A Poet's Musings on Efficient Computation**. Programming Languages Seminar, Indiana University, April 10, 2007.
- **Seven at One Stroke: Results from a Cache-Oblivious Paradigm for Scalable Matrix Algorithms**. Programming Languages Seminar, Indiana University, October 19, 2006.
- **Existential Types: How to write first class patterns in Haskell**. Programming Languages Seminar, Indiana University, September 7, 2006.

Publications

- Michael D. Adams, Andrew W. Keep, Jan Midtgaard, Matthew Might, Arun Chauhan, and R. Kent Dybvig. **Flow-sensitive type recovery in linear-log time**. In *OOPSLA '11: Proceedings of the ACM international conference on Object oriented programming systems languages and applications*, New York, NY, USA, October 2011. ACM. To appear.

– Michael D. Adams. **Scrap your zippers: A generic zipper for heterogeneous types**. In *WGP '10: Proceedings of the 2010 ACM SIGPLAN workshop on Generic programming*, pages 13–24. ACM, New York, NY, USA, 2010. doi: 10.1145/1863495.1863499.

– Andrew W. Keep, Michael D. Adams, Lindsey Kuper, William E. Byrd, and Daniel P. Friedman. **A pattern matcher for miniKanren or how to get into trouble with CPS macros**. In *Scheme '09: Proceedings of the 2009 Scheme and Functional Programming Workshop*, number CPSLO-CSC-09-03 in California Polytechnic State University Technical Report, pages 37–45. 2009. URL `http://digitalcommons.calpoly.edu/csse_fac/83/`.

– Michael D. Adams and R. Kent Dybvig. **Efficient nondestructive equality checking for trees and graphs**. In *ICFP '08: Proceeding of the 13th ACM SIGPLAN international conference on Functional programming*, pages 179–188. ACM, New York, NY, USA, 2008. doi: 10.1145/1411204.1411230.

– Peter Gottschling, David S. Wise, and Michael D. Adams. **Representation-transparent matrix algorithms with scalable performance**. In *ICS '07: Proceedings of the 21st annual international conference on Supercomputing*, pages 116–125. ACM, New York, NY, USA, 2007. doi: 10.1145/1274971.1274989.

– Michael D. Adams and David S. Wise. **Seven at one stroke: Results from a cache-oblivious paradigm for scalable matrix algorithms**. In *MSPC '06: Proceedings of the 2006 workshop on Memory system performance and correctness*, pages 41–50. ACM, New York, NY, USA, 2006. doi: 10.1145/1178597.1178604.

– Michael D. Adams and David S. Wise. **Fast additions on masked integers**. *SIGPLAN Notices*, 41(5):39–45, May 2006. ISSN 0362-1340. doi: 10.1145/1149982.1149987.

– Michael D. Adams. **The representation of constraints, annotations and first class patterns over arbitrary data types in Haskell**. Honors Undergraduate Research, University of Kansas, May 2004.