Type Flow Analysis for Exploratory Software Development

Philip M. Johnson
Ph.D. Dissertation
Computer and Information Science Department
University of Massachusetts

COINS Technical Report 90-64
September 1990

Type Flow Analysis for Exploratory Software Development

A Dissertation Presented

by

Philip M. Johnson

Submitted to the Graduate School of the
University of Massachusetts in partial fulfillment
of the requirements for the degree of

Doctor Of Philosophy

September 1990

Computer and Information Science

TYPE FLOW ANALYSIS FOR EXPLORATORY SOFTWARE DEVELOPMENT

A Dissertation Presented

by

PHILIP M. JOHNSON

Approved as to style and content by:

Victor Lesser, Co-Chair

Jack Wileden, Co-Chair

Wendy Lehnert, Member

David Stemple, Member

Daniel Corkill, Member

John Buonaccorsi, Member

W. Richards Adrion, Department Chair
Computer and Information Science

Acknowledgements

This research results from almost seven years of oscillation between the fields of Artificial Intelligence and Software Engineering. I'm sure many departments would have nipped this ambivalence in the bud; I feel very lucky to have been not only tolerated, but even supported in my search for a research direction related to both disciplines.

My co-chairs, Jack Wileden and Victor Lesser, provided intellectual, emotional, and financial support through thick and thin. Any contributions this research makes pays tribute to their abilities as researchers, mentors, advisors, and teachers.

I thank all of my committee members for their careful and timely reading of my manuscript, and for their insightful comments and corrections. In addition, I thank Wendy Lehnert for being the first faculty member to take a chance on me as a research assistant, and for maintaining interest in and commitment to my work as it swerved from natural language processing toward artificial language processing. When he wasn't delighting me with tales of rare bird sightings, David Stemple helped me understand the finer points of polymorphic type inference. Dan Corkill taught me a lot of what I know about exploratory

software development. Finally, I thank John Buonaccorsi for helping me see how a non-computer scientist reacts to this work.

Seven years is time enough to make old friends out of fellow graduate students. I thank my COINS friends, and especially the comrades of the Thesis Liberation Front, for everything from the Pina Colada recipe to the `string-reverse-search-not-char` function. I am also graced with a network of friendships reaching from the business schools of Boston to the redwoods of California. Their support has been essential.

My mother, father, brother, and sisters are my oldest friends of all, and I thank them for their love, for their support, and for teaching me by example how to reach for the stars. I thank my darling daughter, Jenna Corin Amberg-Johnson, for being the perfect antidote to academia.

Finally, I thank Joanne Amberg for making these last years the happiest ones of my whole life. That's quite an accomplishment, given that I've been writing a dissertation for most of them. This is for you, of course.

Abstract

Type Flow Analysis for Exploratory Software Development

September 1990

Philip M. Johnson, B.S., University of Michigan

M.S., University of Massachusetts

Ph.D., University of Massachusetts

Directed by: Professor Jack Wileden and Professor Victor Lesser

"Exploratory" programming languages and development enjoy a reputation for enabling both rapid development of prototype implementations and ease of evolution as experience with these prototypes suggests improvements. However, the exploratory paradigm also suffers from a reputation for producing brittle systems, prone to errors and run-time failure. The run-time, dynamic typing associated with exploratory languages contributes to both of these reputations. On the one hand, the ability of exploratory programs to define new types during execution and to dispatch based upon the types of data objects facilitates development by providing powerful forms of polymorphism and abstraction. On the other hand, these same capabilities preclude conventional approaches to compile-time assessment of type safety, such as the declaration-based mechanisms in languages like Pascal, or the inference mechanisms in functional languages like ML.

This dissertation describes a program analysis technique called "type flow analysis" that assesses the type safety of programs written in Lisp. Type flow analysis differs from other compile-time assessment approaches in mechanism and in application. Its mechanism comprises a novel synthesis of polymorphic type inference, symbolic execution, and declaration-based approaches. It is applied as part of an incremental, evolutionary approach to the development of verifiably type safe exploratory programs. Type flow analysis improves the exploratory paradigm by enabling a system to smoothly evolve from an initial, "weakly typed" prototype to a more robust, "strongly typed" end product.

Table of Contents

List of Figures

List of Tables

Chapter 1

Overview

1.1 Toward robust exploratory software

This dissertation presents an approach to developing robust exploratory software. Many software engineers react to "robust exploratory software" as if it were an oxymoron—how could programs written in languages like Lisp using no discernible development methodology be even moderately free of errors?

This perception of exploratory software springs to a great extent from the dearth of facilities for *verification*—techniques to assess whether the program behaves correctly or not. More conventional development methods and languages enjoy a spectrum of verification mechanisms missing from their exploratory counterparts.

For example, the lifecycle model of development [Boe76] requires a mostly linear progression through requirements, specifications, design, implementation, testing, and maintenance. This "waterfall" of development forces developers to specify the behavior of the system long before it is implemented. In addition, the pre-implementation phases help direct the implementation effort

by providing standards against which the correctness of the system can be measured. Finally, the gradual step-wise refinement of the system from abstract requirements to specific modules supports structural clarity in the final implementation.

In contrast to the waterfall method, exploratory software is "specified" through the experiences gained from a succession of implementations [She84, San78, Par86]. For example, the correct behavior of a dynamic, graphical, "user-friendly" interface involves qualitative, look-and-feel issues that are problematic to express in a requirements or specification document [SB82]. For such systems, development normally begins with the implementation of an initial prototype of the system, which then evolves in response to feedback from the user. In these exploratory scenarios, requirements, specification, and design documents provide virtually no verification support. In addition, maintaining structural clarity in an evolving implementation is a much more difficult goal than producing structural clarity in a "single" implementation produced through step-wise refinement.

On top of the lack of developmental verification mechanisms, exploratory software development also lacks an important form of language-level support for verification. Non-exploratory languages such as Pascal and Ada are commonly termed *strongly typed*: executable programs in these languages are free of virtually all type-level errors [CW86]. This type-level consistency assessment mechanism is absent from exploratory languages like Lisp—in these languages, it is quite possible for execution to terminate from an attempt to invoke an addition function on a list data object, for example. Thus, ex-

ploratory software may contain a whole class of type-level errors absent from non-exploratory software. The impact of these errors is only exacerbated by the fact that exploratory development requires so much re-working of a running system, and attendant opportunities for making such type-level errors.

There is a method to this madness, of course. By sacrificing the use of the verification mechanisms employed in strongly typed languages, exploratory languages gain powerful facilities for data and control abstraction [AS87]. These facilities are so conducive to the exploratory style of development that many developers pay the price in the loss of verification mechanisms.

This research addresses the verification problem on both the language and the development levels. Its verification support is based upon a program analysis technique called *type flow analysis*, implemented in a computer program called ESSIE[1]. ESSIE can detect many kinds of type-level errors in Lisp programs, providing exploratory development with a language-level verification mechanism similar to the type checking mechanisms in non-exploratory languages.

Type flow analysis also provides verification support on the developmental level. Type flow analysis does not preclude the development of *weakly typed* programs, where not enough type-level information exists in the source code to verify its structural consistency. When ESSIE cannot verify the type-level consistency of a program, it can instead indicate areas of *structural ambiguity*—the functions whose structure cannot be determined to the extent needed to

[1] ESSIE is an acronym for a system supporting "Exploratory Software Structure Inference and Evolution." ESSIE is also a homonym for the acronym of Software Engineering.

guarantee the absence of run-time conflicts. Type flow analysis improves the exploratory development paradigm by providing three options: (1) simply ignore the feedback provided by ESSIE; (2) provide *structural annotations*—similar to type declarations—which help ESSIE's analysis over these rough spots without ruling out the possibility of run-time errors; and (3) rewrite the code to eliminate the structural ambiguity. These three alternatives correspond naturally to early, middle, and late stages of exploratory development.

In early development, when the implementation is highly volatile, the information from type flow analysis might be ignored on the basis that a type-level error-free prototype is not worth the effort required. After the development process "settles down" somewhat, and the structure of the system is much less susceptible to radical revision, an incremental, phased approach to increasing the robustness of the system can begin. In this middle stage of development, ESSIE can be used to simply identify and document (via annotations) the structurally ambiguous parts of the system and the actual structure intended. As the system further matures into late stages, where the structure of the system is well established, developmental priorities can shift from experimentation with different behavior to verifying the type-correctness of the system. At this point, the ambiguous parts of the system would be re-implemented in a way that allows ESSIE to verify their consistency without supplemental annotations.

Type flow analysis for exploratory software development provides a new alternative to the "rapid prototyping" style [Luq89] of development. Rapid prototyping concatenates a preliminary exploratory phase of development with

a later, lifecycle approach. Two problems with rapid prototyping are: (1) the difficulty of deciding when the exploration phase is over, and (2) the developmental overhead incurred by a complete re-implementation of the system from a weakly typed language to a strongly typed language. With type flow analysis, exploratory development can permit a smooth evolution toward type consistency without requiring an abrupt decision to terminate exploration, and without requiring a shift to a different language.

Type flow analysis does not simply consist of adapting the type verification strategies of languages like Pascal, Ada, or even ML to the syntax of Lisp. The differences between these languages do not consist of the simple presence or absence of type verification mechanisms, but consist instead of fundamentally different perceptions of the nature and role of types in a programming language. The next section presents a "look through the keyhole" at these differences, and at the use of type flow analysis in exploratory software development. Following this simple introduction is a survey of the general themes, thesis, and contributions of this research. The final section of this overview chapter provides a guide to the remainder of this dissertation.

1.2 Type flow analysis in action

Exploratory languages such as Lisp provide typing mechanisms that differ in fundamental ways from those in conventional languages like Ada and Pascal, as well as from functional polymorphic languages like ML or Haskell.

For this reason, type flow analysis provides a new representation for type-level structure that differs significantly from that employed for either conventional, declaration-based languages or functional, inference-based languages. This section illustrates the use of this *execution strand*-based representation. As will be shown, type flow analysis for exploratory software development provides many of the same type-level benefits found in conventional or functional languages—the most important being the ability to statically detect many forms of type-level errors. This section also illustrates some limitations of type flow analysis, such as the signalling of "ersatz" type errors.

1.2.1 Expressiveness and brittleness in exploratory software

Figure 1 shows three functions from a simple Lisp program that reveal some of the distinguishing features of exploratory languages. Despite their simplicity and utility, these functions cannot be written in a language like Pascal or Ada. The first function, `size`, simply returns a numeric value corresponding to the size of its argument. `Size` is an interesting function because it accepts arguments of several different types: hash tables, vectors, and arbitrary kinds of lists—for example, a list of integers, a list of strings, a list of lists, and so forth.

`Size` is used by the second function in Figure 1, push. Push is one of a set of functions to implement a stack. It accepts a stack object, and an item to be added to the stack, and returns a new stack object with the added item. The interesting aspect of push is that the passed stack can be either of two different

- Size *returns the length of different types of objects.*

```
(defun size (obj)
  (cond ((listp obj)
         (list-length obj))

        ((hash-table-p obj)
         (hash-table-count obj))

        ((vectorp obj)
         (array-length obj))))
```

- Push *returns a new stack with the item added.*

```
(defun push (item stack)
  (cond ((and (listp stack) (< (size stack) 10))
         (push-list item stack))

        ((and (listp stack) (>= (size stack) 10))
         (push-vector item (list-to-vector stack)))

        ((vectorp stack)
         (push-vector item stack))))
```

- Stack-bottom *returns the current bottom of the stack.*

```
(defun stack-bottom (stack)
  (last stack))
```

Figure 1: Three functions from a simple Lisp program.

types—a list or a vector. Further, if the stack passed is a sufficiently large list, rather than simply adding the item to this list, the function re-implements the stack as a vector and returns this new type of stack with the added item.

`Size` and `push` represent simple, yet powerful forms of abstraction. `Size` provides *genericity*, so that the programmer can call a single function whenever the size of an object is required regardless of its type. `Push` provides *information hiding*: the programmer can add an item to the stack without concern for what kind of stack it is, and the stack itself can change in form over the course of execution. Such language-level expressiveness is rare outside of the exploratory domain.

The type-level structure of the first two functions in Figure 1 is straightforward. `Size` accepts vectors, hash tables, and arbitrary lists, and returns a number. `Push` accepts an item (of arbitrary type), and a stack, (which could be either a list or a vector), and returns either a list or a vector[2]

The third function, `stack-bottom`, is *supposed* to return the last item in a stack. However, `stack-bottom` contains a structural error: it only handles the case where the passed stack is a list. When a stack is dynamically redefined

[2]For the purpose of this introduction, assume that:

1. Lists and vectors are *homogeneous*—storing objects of only one type at a time. Later chapters will explore how type flow analysis supports *heterogeneity*—the ability of Lisp objects such as lists to contain objects of several different types at once.
2. "Falling off the end of a cond clause"—i.e., the failure of any conditional test to succeed—indicates an error rather than an implicit `nil` return value from the cond clause. The pros and cons of this approach are discussed in Chapter 4.

to a vector and then passed to `stack-bottom`, the program will abort with a run-time error.

`Stack-bottom` reveals the price that exploratory programmers have traditionally paid for the abstraction capabilities of their languages. This price is, quite simply, the loss of facilities to verify the structural consistency of their programs through static analysis. In strongly typed languages, type-level inconsistencies of the form exhibited by `stack-bottom` would be detected at compile-time, rather than surviving into the run-time environment. This difference in the time of error-detection is severe: while the original developers are guaranteed to be available and motivated to correct a type-level error during the initial compilation of the system, run-time errors may not occur until months or even years after delivery of the system, depending upon the frequency with which the incorrect code is executed.

Type flow analysis provides a means by which developers can both retain the expressiveness of exploratory languages, yet enjoy compile-time detection of errors such as that exhibited by `stack-bottom`. The next section introduces the representation for type-level structure developed in this research, and the following section illustrates how it is used to uncover errors.

1.2.2 Representing type-level structure with execution strands

Figure 2 illustrates a representation of these functions produced by type flow analysis. This representation of a function is called a *braid*, and each braid is made up of a set of *execution strands*, or strands for short. Braids are

- *Type flow analysis representation for* `size`*:*

$$Br_1 \left[\begin{array}{ll} St_{11} \left\{ \begin{array}{ll} FC: & (listp\ obj) \\ CE: & obj : \mathsf{list}[\mathsf{v}_1] \\ RT: & \mathsf{integer} \end{array} \right. \\ St_{12} \left\{ \begin{array}{ll} FC: & (hash\text{-}table\text{-}p\ obj)\ (not(listp\ obj)) \\ CE: & obj : \mathsf{hashtable}[\mathsf{v}_2] \\ RT: & \mathsf{integer} \end{array} \right. \\ St_{13} \left\{ \begin{array}{ll} FC: & (vectorp\ obj)\ (not(hash\text{-}table\text{-}p\ obj))\ (not(listp\ obj)) \\ CE: & obj : \mathsf{vector}[\mathsf{v}_3] \\ RT: & \mathsf{integer} \end{array} \right. \end{array} \right]$$

- *Type flow analysis representation for* `push`*:*

$$Br_2 \left[\begin{array}{ll} St_{21} \left\{ \begin{array}{ll} FC: & (listp\ stack) \\ CE: & item : \mathsf{v}_1; stack : \mathsf{list}[\mathsf{v}_1] \\ RT: & \mathsf{list}[\mathsf{v}_1] \end{array} \right. \\ St_{22} \left\{ \begin{array}{ll} FC: & (listp\ stack) \\ CE: & item : \mathsf{v}_2; stack : \mathsf{list}[\mathsf{v}_2] \\ RT: & \mathsf{vector}[\mathsf{v}_2] \end{array} \right. \\ St_{23} \left\{ \begin{array}{ll} FC: & (vectorp\ stack) \\ CE: & item : \mathsf{v}_3; stack : \mathsf{vector}[\mathsf{v}_3] \\ RT: & \mathsf{vector}[\mathsf{v}_3] \end{array} \right. \end{array} \right]$$

- *Type flow analysis representation for* `stack-bottom`*:*

$$Br_3 \left[St_{31} \left\{ \begin{array}{ll} FC: & T \\ CE: & stack : \mathsf{list}[\mathsf{v}_4] \\ RT: & \mathsf{v}_4 \end{array} \right. \right]$$

Figure 2: The type flow analysis for three example functions

denoted by Br_1, Br_2, etc., and enclosed by "[" and "]". Strands are denoted by St_1, St_2, etc., and enclosed by "{" and "}". For a fuller explanation of this notation, see Section 3.2.1.

Each strand represents a distinct execution path through the function at the type level, and has three parts: a *feasibility condition*, a *constraint environment*, and a *result type*.

Feasibility conditions (FC) represent the type-related constraints that must be satisfied in order for the execution path represented by the strand to be valid. Feasibility conditions are normally produced from the presence of *type predicates* (i.e. `integerp` or `listp`) in the test clauses of conditional statements. Feasibility conditions are used to determine if one execution path can follow another during the actual execution of the program. If the composition of the two conditions from these paths is infeasible, then the proposed composite execution path cannot occur in reality, and no type-level inconsistency is indicated.

The constraint environment (CE) holds the type-level properties of variables that must be satisfied whenever the execution path is traversed. If the composition of two execution paths is found to be *feasible*, but the two constraint environments cannot be successfully composed, then a type-level error *is* indicated.

Finally, the result type (RT) indicates the end result of executing the particular strand—what type would be returned from the execution of the code represented by the strand.

1.2.3 Uncovering type-level errors with type flow analysis

Braids can uncover structural inconsistencies such as the one represented by `stack-bottom`. Figure 3 illustrates such a scenario: in this code fragment, `stack-bottom` is called on the results returned from a call to `push`. While the precise mechanisms will be described in Chapter 3, the general idea is this: first, the braid corresponding to *push* is retrieved and analyzed to determine the range of result types possible in this context. In this case, since there are no pre-existing constraints on the type of the stack, there are two possible result types: a list, and a vector. Composition of either of the path conditions associated with `push` with the path condition from `stack-bottom` does not lead to an inconsistency, so to maintain structural consistency, `stack-bottom` must be able to accept both lists and vectors. Since `stack-bottom` only accepts list objects, a structural error is signalled.

Type flow analysis, as described thus far, might appear to provide Lisp with all of the type checking properties of languages like Pascal (or more appropriately, ML, since the information is inferred). Like Pascal or ML, it produces a "type" for a function, albeit in an unorthodox format. Most importantly though, type-level inconsistencies are uncovered through static analysis: the problem with `stack-bottom` is uncovered at compile-time, rather than escaping undetected into the run-time environment. Type flow analysis appears to overcome one of the major causes of brittleness in exploratory languages, and would remove a major obstacle to the use of Lisp in applications where robustness and reliability are of paramount importance.

- *Lisp expression to be analyzed:* `(stack-bottom (push item stack))`
- *Assume no prior constraints on* `item` *and* `stack`.
- *The type flow analysis of* `(push item stack)` *is:*

$$Br_2 \left[\begin{array}{ll} St_{21} \left\{ \begin{array}{ll} FC: & (\textit{listp stack}) \\ CE: & \textit{item}: \mathsf{v}_1; \textit{stack}: \mathsf{list}[\mathsf{v}_1] \\ RT: & \mathsf{list}[\mathsf{v}_1] \end{array} \right\} \\ St_{22} \left\{ \begin{array}{ll} FC: & (\textit{listp stack}) \\ CE: & \textit{item}: \mathsf{v}_2; \textit{stack}: \mathsf{list}[\mathsf{v}_2] \\ RT: & \mathsf{vector}[\mathsf{v}_2] \end{array} \right\} \\ St_{23} \left\{ \begin{array}{ll} FC: & (\textit{vectorp stack}) \\ CE: & \textit{item}: \mathsf{v}_3; \textit{stack}: \mathsf{vector}[\mathsf{v}_3] \\ RT: & \mathsf{vector}[\mathsf{v}_3] \end{array} \right\} \end{array} \right]$$

- *However, the type flow analysis of* `stack-bottom` *is:*

$$Br_3 \left[St_{31} \left\{ \begin{array}{ll} FC: & T \\ CE: & \textit{stack}: \mathsf{list}[\mathsf{v}_4] \\ RT: & \mathsf{v}_4 \end{array} \right\} \right]$$

- *While the composition of both strand St_{21} and St_{22} is feasible and consistent with the strand St_{31}, composing strand St_{23} with the strand St_{31} is feasible but* inconsistent, *which signals a type error.*

In normal language, the third case represents push *returning a* vector *and passing it to* `stack-bottom`, *which requires a* list.

Figure 3: Uncovering type errors with type flow analysis

1.2.4 The problem of "ersatz" structural errors.

Unfortunately, this appearance is only partially accurate. While type flow analysis is actually more powerful than the simple examples above indicate, it is still not powerful enough to type check any arbitrary Lisp program—in effect, to make Lisp into a "strongly typed" language. Figure 4 illustrates an example of the limitations of type flow analysis. In this simple expression, an item is pushed on to a stack of type list, and then the first element of that list is retrieved. Such an expression is structurally and semantically valid: pushing one element onto a list stack will return a list, which is a valid argument to the function `first`. However, type flow analysis would signal a structural error in this situation, since it must assume that a vector could be returned from the function.

Even this ersatz structural error has value, however. It points the programmer's attention to the fact that, while perhaps technically correct, it is stylistically suspect to call `first` on a stack object which can be a list or a vector, even though in this particular context the stack can only be a list. Upon reflection, the programmer might very well implement the function `top` illustrated in Figure 5, and use it instead of `first`. This restructuring eliminates the signaling of a structural error, since the expression now accounts for both the list and the vector implementation of stacks.

In conclusion, this simple example illustrates strengths and weaknesses of both exploratory languages like Lisp, and of the type flow analysis mechanism itself. Lisp supports software development through powerful capabilities for

- *The expression* `(first (push 'x nil))` *results in an "ersatz" type error.*
- *The two feasible strands from calling* push *with a list stack are* St_{21} *and* St_{22}. *Strand* St_{23} *is infeasible since it requires* push *to be invoked with a* vector *stack.*

$$Br_2 \left[\begin{array}{ll} St_{21} \left\{ \begin{array}{ll} FC: & (\textit{listp stack}) \\ CE: & \textit{item}: v_1; \textit{stack}: \text{list}[v_1] \\ RT: & \text{list}[v_1] \end{array} \right. \\ St_{22} \left\{ \begin{array}{ll} FC: & (\textit{listp stack}) \\ CE: & \textit{item}: v_2; \textit{stack}: \text{list}[v_2] \\ RT: & \text{vector}[v_2] \end{array} \right. \end{array} \right]$$

- *This indicates that* `(push x nil)` *might return a* vector.
- *However,* `first` *requires a list, so a type error is raised.*
- *The* value-level *semantics of* push *reveal that* `(push 'x nil)` *always returns a list.*

Figure 4: Type flow analysis sometimes signals "ersatz" errors.

- *The* `top` *function eliminates the ersatz error in Figure 4:*

```
(defun top (stack)
  (cond ((listp stack)
         (first stack))
        ((vectorp stack)
         (aref stack (aref stack 0)))))
```

Figure 5: Eliminating ersatz errors through code restructuring

genericity and abstraction. Unfortunately, the mechanism by which this power is obtained has a high cost: the loss of the static type-level verification mechanisms prized in conventional languages and development methods. Type flow analysis does not completely reclaim this capability for exploratory development, in the sense of enabling the type-level verification of every legal Lisp program. Instead, it provides type-level verification for Lisp programs where sufficient static information exists, and pinpoints areas of structural ambiguity in other programs. This latter information often suggests how the software might evolve from a structurally ambiguous state to a verifiably consistent state.

1.3 Research themes

Type flow analysis for exploratory software development provides an alternative perspective to several commonly held notions about the nature of types in programming languages. It also represents a rather unorthodox research direction in type inference techniques. Finally, the mechanisms of type flow analysis represent a novel marriage of program analysis techniques. These general themes can be summarized in the following statements, each of which is subsequently amplified below.

- Programs, not languages, are "strongly" or "weakly" typed.
- Weakly typed languages have significant expressive strengths.

- Type flow analysis adapts type verification to the language, rather than the language to the type verification mechanisms.
- Exploratory languages differ from non-exploratory languages in the nature and role of types.
- Type flow analysis represents the type-level structure of exploratory systems by combining aspects of polymorphic type inference and symbolic execution.

1.3.1 Programs, not languages, are "strongly" or "weakly" typed.

Type-level verification in programming languages is very strongly polarized between the so-called "strongly typed" and "weakly typed" languages. In strongly typed languages (such as Ada, Pascal, or ML), every legally expressible program (i.e., every program that can be compiled) is verified to be free of any type inconsistencies. Weakly typed languages (such as Lisp) are basically the opposite: statically-based type-level verification is minimal or non-existent.

One theme of this research is that the presence of enough type-level information to verify structural consistency should be a property of a particular program, not a programming language. So-called weakly typed languages simply allow the programmer the flexibility to write both statically verifiable type consistent programs, as well as programs whose structure cannot be statically

verified. Type flow analysis demonstrates how to apply verification mechanisms on a program, rather than language, level.

1.3.2 Weakly typed languages have significant expressive strengths.

Even given that some partial form of verification can be provided to exploratory languages like Lisp, the following question might remain: "Why bother?" In other words, why sacrifice the ability to guarantee that every legal program will automatically be free of an important class of programming errors[3].

While exploratory languages sacrifice verification mechanisms, they gain the ability to express important and powerful abstractions. For example, exploratory languages can express a broader spectrum of polymorphism than any current strongly typed language. In addition, exploratory languages allow program structures like types and functions to be defined at run-time, by the executing program itself. This supports a form of procedural abstraction where the programmer no longer writes a program to solve a problem, but rather writes a program that will in turn write another program to solve the problem. The run-time nature of types and other structural entities in exploratory languages provides similar benefits in programming environments, where programs manipulate other programs.

[3]The terms "strongly" and "weakly" typed even have a strong emotional connotation—for example, "strong" is associated with: healthy, sane, lusty, valid, and convincing, while "weak" is associated with futile, vapid, cadaverous, unbelievable, and inadequate. Perhaps such subliminal messages explain the reticence of *real* programmers to use exploratory languages...

1.3.3 Type flow analysis adapts type verification to the language, rather than the language to the type verification mechanisms.

Certain features of exploratory languages, such as polymorphism, are more or less accepted as desirable properties of any programming language. While missing from most strongly typed languages in the 1970's, virtually every recent attempt to bridge this gulf involves the creation of a new, strongly typed language that exhibits incrementally more support for a subset of the facilities traditionally present in exploratory languages.

Type flow analysis takes the opposite tack on bridging this gap: rather than attempting to imbue a strongly typed language with the desirable features of exploratory languages, type flow analysis imbues exploratory languages with some of the desirable properties of strongly typed languages. Both directions involve compromises: just as modern strongly typed languages do not provide all the capabilities of exploratory languages, type flow analysis does not provide the complete verification facilities available in strongly typed languages.

1.3.4 Exploratory languages have profoundly different notions of the nature and role of types.

The fact that type flow analysis fails to make an exploratory language such as Lisp "strongly typed" is not fundamentally due to the mechanism, but to the different nature of types in strong and weak systems. Strongly typed languages uniformly base their verification systems on the following properties of the type system:

1. Sufficient type-level information must be present in the source code in order to verify the structural consistency of the program.

2. Type-level information is *identifier-based.* Type verification consists of ascertaining the type associated with each identifier in the program, then ensuring that each context the identifier appears in is consistent with this type.

3. Type-level information essentially only "exists" in the source code. After the compilation mechanism verifies type-level consistency, object code normally performs no assessment of the types of objects being manipulated. (Numeric coercion mechanisms constitute a "hard-wired" exception to this rule.)

Exploratory languages such as Lisp take a diametrically opposite stance on each of these fronts, which effectively hamstrings the application of conventional type verification techniques to the language:

1. Types and other structural objects can be defined at run-time, and legal (and correct) source code can be ambiguous with respect to its type-level structure. There is no a priori requirement that the source code contain enough information to verify type consistency.

2. Type-level information is *object-based.* Type information is attached to run-time data objects, not to compile-time source code identifiers. Identifiers are not constrained to a single type (or type expression) as they are in strongly typed languages.

3. Type-level information persists into the run-time environment. Exploratory programs compute the type of objects at run-time just as strongly typed languages compute the value of objects.

1.3.5 Type flow analysis represents the type-level structure of exploratory systems by combining aspects of polymorphic type inference and symbolic execution.

In strongly typed languages, verification at the type-level consists of ensuring that the type associated with an identifier is consistent with all the contexts in which that identifier appears. While this form of verification catches many errors, it misses many others that depend upon an improper use of the *value*, rather than the type bound to the identifier.

Symbolic execution [MJ81b, Kin76] is a testing paradigm that attempts to rid implementations of certain classes of these *value-based* errors. In effect, symbolic execution creates a model of the control and data flow in the program, developing symbolic expressions for the *values* associated with identifiers.

For example, consider the following program fragment:

```
(if (< x 5)
    (foo x)
    (bar x))
```

The symbolic execution paradigm derives the fact that when `foo` is called in this context, its argument will be less than 5, while `bar`'s argument will be greater than or equal to 5. When successful, symbolic execution provides

a representation of the different execution paths in the program, as well as a symbolic representation of the values of each occurrence of the identifier along each of these execution paths. Symbolic execution may fail, however, by being unable to successfully synthesize a symbolic expression representing the value of an identifier.

Polymorphic type inference [Car87] can be viewed as a close cousin of symbolic execution: it also performs a control and data flow analysis of a program, but associates a symbolic expression with each identifier representing its type, rather than its value. Furthermore, this type is invariant across all of its occurrences: in the example program fragment above, each occurrence of `x` must be associated with the exact same symbolic expression. Not only that, but the symbolic expressions for the types returned by the expressions `(foo x)` and `(bar x)` must be exactly the same. In return for these restrictions, languages based upon polymorphic type inference gain *completeness*: failure of the type inference algorithm indicates an incorrect program. In contrast, failure of symbolic execution systems merely indicates a complex, though not necessarily incorrect program.

As the example scenario involving push illustrates, Lisp programs violate a principal axiom of polymorphic type inference—that the type of an identifier in each of its usage contexts (or the type returned by each arm of a conditional) be the same. Type flow analysis explores the ramifications of combining selected symbolic execution techniques with others from polymorphic type inference. In type flow analysis, identifiers are not required to take on the same type in each occurrence, nor are conditional arms required to return the same type[4]. In

[4]Type flow analysis shares some of the limitations of symbolic execution with respect to loop analysis, however. In most cases, the set of execution paths representing a loop must

common with polymorphic type inference, however, is a reliance on *unification* as a principal means to detect inconsistencies in the program.

The result of type flow analysis of a function is not a set of type expressions for each of the parameters and its result, as in polymorphic type inference. Instead, it is a set of expressions representing the distinct execution paths for the function at the type level. This execution path representation, as illustrated in the previous section, can be used to detect type inconsistencies just as polymorphic type inference does.

Representing the structure of a function as its execution paths at the type level has both advantages and drawbacks in comparison to polymorphic type inference. The principal advantage is applicability: type flow analysis works on exploratory languages like Lisp, polymorphic type inference does not. However, the drawback is incompleteness: failure of type flow analysis does not necessarily indicate an incorrect program.

1.4 Research Thesis

The research themes presented in the previous section can be distilled down to the following thesis, or claim, about the research presented in this dissertation:

> *Exploratory software development employs type-level constructs that simultaneously provide powerful facilities for abstraction and computational expression, yet preclude application of the type veri-*

---

be independent of the number of loop iterations. Chapters 3 and 5 discuss this in more detail.

fication mechanisms present in strongly typed languages. This contributes significantly to the brittleness of exploratory software.

Type flow analysis provides mechanisms to increase the robustness of exploratory software. It accomplishes this in a manner compatible with both the developmental and language-level nature of the exploratory paradigm.

In arguing for this thesis, research was performed that led to a set of contributions pertaining to type-level analysis and software development. These contributions are summarized in the next section.

1.5 Research Contributions

This research discovered a novel form of representation for the type-level structure of exploratory software in terms of *execution strands*, which represent the set of execution paths through an exploratory function at the type level. Execution strands embody a novel combination of features from symbolic execution systems and polymorphic type inference systems. Entirely new methods for the representation of the three principal forms of computation–sequencing, conditionals, and iteration—for this type-level representation were designed.

Execution strands provide a new and powerful mechanism for the *inference* of the type of polymorphic functions. Through the novel exploitation of the semantics of the type predicate language construct, type flow analysis can infer and represent ad hoc as well as parametric polymorphism. No other type

analysis mechanism can even declare, much less infer, ad hoc polymorphism with the degree of generality supported by this research.

While type flow analysis currently implements a type expression language similar to that employed by parametric polymorphic languages, it synergistically combines with the execution strand machinery to provide new support for the type-level representation of *heterogeneous* data objects. This representation, while still an approximation to the precise structure of the heterogeneous object, is nevertheless a significant advance over other typing mechanisms that prohibit such objects entirely.

To test the ideas of type flow analysis, the ESSIE system was implemented. ESSIE comprises a contribution as a new language for the design and implementation of type analysis systems. The control and data abstractions provided by ESSIE illuminate the requisite properties of control and data flow mechanisms in type analysis, as well as supporting the incremental construction, extension, and experimentation with new forms of type analysis.

An experimental type analysis system was implemented in ESSIE and used to analyze the type-level structure of a small scale exploratory software system. Among other effects, this analysis uncovered a previously unknown type-level error, elucidated previously unknown structural capabilities of a function, and uncovered a previously unrecognized subtle structural brittleness that led directly to the restructuring of the software into a more robust form.

Finally, a large scale exploratory software system was studied for evidence that the kinds of representations provided by ESSIE are indeed necessary for

the structural analysis of "real world" exploratory software. The results support the direction taken by type flow analysis: at least 40% of the polymorphism in the particular system studied is of the ad hoc nature focused upon in this research.

1.6 Organization of the dissertation

This section provides a brief summary of the remaining chapters in this dissertation. Fortunately, depending upon the interests and backgrounds of the reader, various parts of this dissertation can be skipped with impunity. This section concludes with some examples of how to successfully accomplish such an abridgement.

Chapter 2 presents an overview of the prior research related to this dissertation. It begins by placing type flow analysis for exploratory software development within the general context of the subdiscipline of Artificial Intelligence and Software Engineering. It next reviews the concept of *exploratory software development* from several perspectives, providing another angle on the motivation for the approach taken in this research. Chapter 2 then turns to the technical aspects of symbolic execution and polymorphic type inference, comparing and contrasting these techniques to those of type flow analysis.

Chapter 3 presents the mechanisms of type flow analysis: the fundamental ways in which the type-level structure of exploratory software is inferred and manipulated. In this chapter, the exposition of both the target language

and the analysis mechanisms are presented in a relatively implementation and domain-independent manner. It presents the three fundamental processes of control and data flow for type flow analysis: sequential processing, conditional processing, and iteration.

Chapter 4 takes the fundamental concepts of Chapter 3 and brings them into the real world by describing the language of type flow analysis. This chapter presents an overview of ESSIE, an implementation of the mechanisms of type flow analysis. ESSIE's architecture implements not simply a single type analysis system, but rather a new language for the description of a class of type analysis systems based upon the mechanisms of type flow analysis. Chapter 4 describes this language and provides an overview of its use in the creation of an actual type flow analysis system.

Chapter 5 presents an evaluation of type flow analysis. First, it presents two studies from the literature supporting the validity of the direction taken in this research: that combining the language and developmental paradigms of exploratory software with the robustness provided by strongly typed languages would solve an important problem in software engineering. Second, the chapter presents the results of applying an experimental type analysis system developed in ESSIE to a small scale exploratory software system.[5] This study reveals many interesting capabilities and limitations of the mechanisms of type flow analysis as they stand now, and points toward promising avenues for fu-

[5]These results are elaborated in the Appendices.

ture research. The final part of Chapter 5 presents a "type-level analysis[6]" of a large scale exploratory software system. The results of this study affirm the basic motivations and assumptions underlying the direction taken in this research.

Chapter 6 concludes this dissertation with a summary of the current status of type flow analysis, the contributions made thus far, and some promising directions for future research.

Readers with backgrounds in either exploratory software development, symbolic execution systems, or polymorphic type inference techniques may skip the associated sections of Chapter 2 without loss of continuity. Most readers should peruse the first four sections of Chapter 3, which provide an overview of the representations and mechanisms of type flow analysis, but the subsequent sections detailing the processing of sequential statements, conditionals, and iteration may be skipped on the first reading. In Chapter 4, the first language construct presented in each major section (i.e. **define-symbol-analysis**, **analyze-expression**, **define-analysis**, and **define-analyzer**) is relatively essential to understanding the language of type flow analysis, while the remainder are somewhat less central. The most important part of the evaluation presented in Chapter 5 is that of ESSIE itself, and is mandatory reading. Finally, the future directions provide an important sense for the continuous and ongoing nature of this research.

---

[6]Rather than a "type flow analysis", which would have necessitated a prohibitive scaling up of the experimental type analysis system.

Chapter 2

Related Research

2.1 Type Flow Analysis as AI and SE

The concerns addressed by type flow analysis for exploratory software development span a number of research disciplines—programming languages, type disciplines, inference techniques, and development methods. While this dissertation does not occupy any single niche comfortably, its broadest categorization is within the research subdiscipline termed *artificial intelligence and software engineering*, or AI and SE.

Though the definition and boundaries of the subdiscipline of AI and SE are difficult to discern, research interest in it is not, as evidenced by the growing body of paper compendia [RW86, Par90], special journal issues [Te88a, Te88b], and survey/overview articles [Mos85, Par86, For87, LD89, Bar87, Sim86].

In general, research in AI and SE consists of technology transfer from one discipline to the other. Most commonly, this takes the form of applying AI concepts such as heuristic search, knowledge representation, theorem proving, and expert systems techniques to the domain of software development. Ex-

amples of these kinds of research can be found in the overview literature cited above.

A more implicit form of technology transfer is resulting from the assimilation of AI exploratory programming techniques into the SE mainstream. This methodological cross-fertilization will be discussed further below.

Type flow analysis for exploratory software development fits in AI and SE by transfering technology between the two disciplines. However, this transfer runs against the tide of the above research by applying SE techniques to AI. As introduced in the first chapter, type flow analysis uses polymorphic type inference and symbolic execution techniques to form a type-level representation of exploratory software, which aids developers in producing robust systems.

From this perspective, this research follows the direction advocated by Winograd in a pair of seminal articles on the future of software development. [Win73] stresses the importance of programming environments that provide explicit, detailed models of the programs under development within the environment. In [Win79], he suggests that "The main goal of a programming system should be to provide a uniform framework for the information that now appears (if at all) in the declarations, assertions, and documentation. The system should provide a set of tools for generating, manipulating, and integrating descriptions both of results and processes." Type flow analysis accomplishes precisely this goal for type-level information in exploratory software.

The next section describes how type flow analysis provides a unique manner of addressing important problems relating to the robustness of exploratory

software development. After this motivational section for the techniques of type flow analysis, the following two sections provide a context for these techniques within the general areas of flow analysis and type systems.

2.2 Addressing the Brittleness of Exploratory Software Development

2.2.1 Exploratory development and the waterfall model

The term *exploratory software development* was coined in 1984 by Beau Sheil [She84] to refer to the "conscious intertwining of system design and implementation." This perspective is re-iterated in [BS86]: "specification and implementation evolve together as the program is understood and tested" and in other research, such as [Gol87]. Though the term was new, the paradigm had been emerging over the previous decade as AI practitioners became aware that this *structured growth* of software [San77] differed qualitatively from the development technique then in vogue with software engineers. This latter method was the *waterfall lifecycle model.*

The description of the waterfall lifecycle in [Ost81] provides a particularly interesting contrast to exploratory software development:

- *Requirements Definition.* The phase during which the basic needs of the software project are enunciated. These needs are to be specified functionally, not algorithmically.

- *Preliminary Design.* This phase explores possible strategies for building an algorithmic solution to satisfy the needs of the requirements. Preliminary design work may lead to refinement or changes in the requirements. Osterweil notes that "the requirements and preliminary design activities should be viewed as iterative and intertwined. Together they should be considered to be the process of gaining understanding of the nature of the problem and an acceptable approach to its solution."

- *Detailed Design.* This phase elaborates the solution approach discovered during preliminary design down to code-level specifications. Osterweil continues: "Detailed design should not be viewed as merely an extension of the preliminary design activity. At the start of detailed design it is necessary for the designers to reorient their thinking from a problem-understanding orientation to a software construction orientation."

- *Coding.* This phase involves what should be a rather straightforward translation of the detailed design into actual source code.

Both exploratory and waterfall development require intertwining—they simply differ in *what* is intertwined. By intertwining requirements and preliminary design, waterfall development prescribes a dialogue between the user and the system analyst during the process of design. However, as Osterweil emphasizes in his description of detailed design, this process strictly precedes the implementation of the system: it is a process-level version of the code-level "fire walls" described in [She84]. Exploratory programming breaks down this

fire wall, explicitly incorporating the process and products of the coding phase into the preliminary and detailed design phases.

[SB82] claims that the intertwining of specification and implementation is inevitable when hardware platforms or user requirements are susceptible to change. These ideas are restated in [MJ81a] through the analogy of a supermarket:

> What we understand to be the [waterfall] approach might be compared with a supermarket at which the customer is forced to provide a complete order to the stock clerk at the door to the store, with no opportunity to roam the aisles—comparing prices, remembering items not on his list, or getting a headache and changing his mind about what to have for dinner. Such restricted shopping is certainly possible and sometimes desirable—it's called mail order—but why should anyone wish to impose that restricted structure on all shopping?

When specification/design and implementation are intertwined, the *readability* of the implementation becomes an important issue, since the implementation serves as an integral part of the specification [Gol87]. Readability goes beyond the *appearance* of the source code—the indentation, type faces, mnemonic identifiers, and so forth—to include the capabilities of the environment to analyze and answer questions about the developing design/implementation. Window-based browsers, inspectors, cross-referencers,

and trace facilities are an integral part of exploratory programming environments (see [BSS84] for examples).

The importance of the environment may go beyond the support of readability, since the programmer is often given access to the implementation of the environment itself [Gol87]. In this case, *environment* as well as system readability is important, and the distinction between the system under development and the environment (which itself becomes an instance of an exploratory system) begins to blur. [Goo81] describes some of the type-level implications of this merging of the development environment with the developing system. He argues that these situations require the dynamic typing capabilities of exploratory languages, in particular, the ability to define types at run-time. Without this capability, the environment cannot use the type-level facilities of its own implementation language to represent the types in the development system, but must programmatically construct its own version of a type system.

Exploratory development thus differs significantly from the waterfall method in intertwining specification and implementation, and this results in new demands upon and relationships between the system under development and the system used for development. The next section constrasts exploratory development with a more recent (and more similar) lifecycle model called "rapid prototyping."

2.2.2 Exploratory development and rapid prototyping

While the intertwining of specification and implementation was originally sufficient to distinguish exploratory programming from traditional SE development methods, a great deal of current SE research involves modifying or replacing the waterfall model with development methods incorporating this property. These development methods are usually termed *rapid prototyping*. Most rapid prototyping techniques are *iterative* rather than *evolutionary*: the prototype(s) are thrown away once they achieve their purpose in obtaining the user's requirements and entailed specifications [Luq89]. In contrast, evolutionary rapid prototyping [CS89] allows the prototype to evolve into the delivered system, and such systems as DEMETER [LR88] form a modern day equivalent of Sandewall's structured growth.

Evolutionary rapid prototyping and exploratory programming have essentially equivalent views of the development process. What distinguishes exploratory programming from evolutionary rapid prototyping is the language and environment features deemed important for facilitating this evolution.

Exploratory programming languages and environments have the following two characteristics: they minimize the amount of *code inertia*, and they support *control* as well as *data* abstraction. Code inertia occurs through language-level redundancy, such as type declarations, which require widespread modifications to the code for relatively simple behavioral changes (for example, changing the type of a variable from an integer to a real in every procedure where the variable is passed.) [She84] cites the declaration-less nature of ex-

ploratory languages like Lisp as important in reducing code inertia and the resultant system "fire walls" that impede evolution.

The capability of the implementation language to express control as well as data abstraction is important in exploratory programming since it frequently involves the "design of experimental specialized sublanguages that make it easier to express solutions to the problems being attacked"[BS86]. This idea is elaborated in [AS87]:

> [In conventional top-down development], a system is designed as a predetermined combination of parts that have been carefully specified to be combined as determined. Each of the parts is itself designed separately by this process. The methodology is flawed: if a system is to be robust, it must have more generality than is needed for the particular application. The means for combining the parts must allow for after-the-fact changes in the design plan as bugs are discovered and as requirements change.
>
> To this end expert engineers stratify complex designs. Each level is constructed as a stylized combination of interchangable parts that are regarded as primitive at that level. The parts constructed at each level are used as primitives at the next level.

Abelson and Sussman note that while conventional languages like Ada or Pascal support data abstraction, they contain restrictions that prevent control abstraction, severely limiting the ability of the designer to stratify the design. These restrictions are:

1. Requiring that a procedure be named and then referred to by name, rather than stating its definition at the point of reference.

2. Forbidding procedures to be returned as the value of other procedures.

3. Forbidding procedures to be components of such data structures as records or arrays.

2.2.3 Type flow analysis improves exploratory development

From this review of the literature, the following three components emerge as essential properties of the exploratory paradigm:

1. A method of development in which experimentation with an evolving implementation forms an essential component of the requirements and specification process. Moreover, the very existence of this implementation may, in some cases, change the problem domain itself, thus leading to changed requirements [Gid84].

2. An environment that supports the use of the implementation as a form of requirements and specifications.

3. An implementation language that minimizes the resistance of the source code to change, and that supports both data and control abstraction.

Type flow analysis for exploratory software development focusses on the development of mechanisms for type consistency verification as a means to

remove errors, and thus lessen the brittleness of exploratory software. This is accomplished in a manner consonant with the development process:

- Code redundancy and inertia are minimized by relying on type inference mechanisms.
- The type-level documentation provided by the analysis improves the readability of the software.
- The analysis mechanism does not place a priori restrictions on the range of control and data abstraction constructs.
- The analysis mechanism integrates smoothly with an incremental, evolutionary approach to development. The analysis can be applied to a single function or an entire system, and the frequency of its application is left to the judgement of the developer.

These four properties of type flow analysis distinguish it from the two other approaches to increasing the robustness of exploratory software—the language-based approach and the development method-based approach.

In the language-based approach, the brittleness of exploratory software is addressed by using a strongly-typed language that nevertheless provides some of the properties for exploratory development missing from languages like COBOL or FORTRAN. For example, languages like Ada and Pascal introduce better forms of data abstraction than those present in COBOL or FORTRAN. However, they remain inappropriate for exploratory software development be-

cause they lack the mechanisms for control abstraction, and because their type declarations introduce inertia.

Languages like ML [GMW79], Miranda [Tur85], and Haskell [HWA+90] lay a much stronger claim to adequacy for exploratory development. Like weakly typed languages, they do not require type declarations[1] yet they uncover all type-level errors through inference. In addition, they support some form of control abstraction, as well as certain forms of polymorphism. Although the generality of control abstraction and polymorphism provided in these languages is significantly less than that provided by weakly typed languages like Lisp, it is fair to say that languages like Haskell do present a valid alternative approach to resolving the type-level brittleness associated with exploratory software development. Type flow analysis, on the other hand, eschews the restrictions on control abstraction and polymorphism required in Haskell, yet trades off the power of the inference technique to detect any, all, and only type-level errors.

The development method-based approaches tend not to embrace the evolutionary refinement from an initial, prototypical implementation to a mature, production-quality final implementation. Most rapid prototyping restricts the exploratory development phase to an initial period, which is then followed by a conventional, waterfall style of development. The spiral model allows the implementation to be re-implemented at each "turn of the wheel." Type flow analysis is alone in explicitly supporting the evolutionary change from a brit-

[1] Although Haskell does require user-specified type information in certain circumstances.

tle, weakly-typed initial implementation to a more robust, strongly-typed final product.

The next two sections turn from the research relating to the motivation for type flow analysis to the research relating to the techniques underlying type flow analysis. The next section contrasts type flow analysis to other kinds of flow analysis. Following this, type flow analysis is compared to type inference.

2.3 Type flow analysis as flow analysis

As the name suggests, type flow analysis is a specialized instance of the more general paradigm known as *flow analysis* [MJ81b]:

> Flow analysis is a tool for discovering properties of the run-time behavior of a program without actually running it. The properties discovered usually apply to all possible sequences of control and data flow, and so give global information impossible to obtain by individual runs or by inspection of only a part of a program. Frequently flow analysis can be viewed as executing the program in parallel over a symbolic, much-simplified version of its real data domain and hence is known alternately by the names symbolic execution or abstract interpretation.

Two survey articles related to flow analysis appear in [FO76, RP86], and [Hec77, MJ81b] contain book-length overviews of the subject. Some of the

problems flow analysis techniques have been applied to are: compiler optimization [Ten74, Bee88]; program verification [CC77]; automatic programming systems [RS76]; and testing [CR81].

From the standpoint of flow analysis, type flow analysis addresses the problem of the symbolic execution at the type level of programs written in a polymorphic, higher-order programming language. This differentiates it from other flow analysis research, which concerns symbolic execution at the value level, as well as from type-level symbolic execution, which does not address polymorphism and/or higher order issues to the same extent[2].

2.3.1 Value-level symbolic execution

Symbolic execution concerns "the execution of the program with symbolic values for each variable whose value is indeterminate at compile-time. [MJ81b]" This results in a representation of the set of paths through the program, and expressions representing the value of each variable at the conclusion of execution. Typically, each variable in the program is initialized to a symbolic name, and the program is then incrementally, symbolically interpreted over these symbols rather than actual input values. The result is a set of algebraic expressions representing the final values of the variables in terms of their initial symbols. While obtaining a completely accurate symbolic representation of an arbitrary program is undecidable, even somewhat imprecise

[2]Note that "symbolic execution at the type level of programs written in a polymorphic, higher-order programming language" is precisely the goal of polymorphic type inference research, reviewed in an upcoming section. Different issues, such as parametric and ad-hoc polymorphism, will be used to differentiate type flow analysis from this line of research.

representations can support useful analyses in areas ranging from program testing to compiler optimization. To provide a flavor for the idea, the issues involved with conditional branches of control and iteration are briefly described below.

Conditional statements represent branches of control, and require the symbolic execution system to augment the symbolic representation of the values of variables with a *path condition*, a predicate representing the conditions under which the path can actually be executed in practice. Consider the following conditional statement:

```
(if (>= x 0)
    (+ x 5)
    (/ 5 x))
```

The first conditional branch will only be executed under the path condition `(>= x 0)`, whereas the second requires `(< x 0)`. Such conditions are necessary, for example, for a program analysis system to ensure that no divide-by-zero errors occur in the program. Additional conditional statements along the execution path lead to the conjunction of the path conditions extracted from the conditional test expressions. The joining of control flow at the end of conditional statements is represented by the disjunction of the path conditions associated with the merging paths.

Iterative statements, like conditionals, require the joining of control flow, but unlike conditionals, the number of joins is potentially unbounded. In some cases, this doesn't matter, particularly if the symbolic values of variables converge to a fixed point after some number of iterations. However, as described

in [Hec77], other loops are "non-stabilizing" with respect to the values of variables. For example, consider a loop containing the sequence of statements:

```
(setf X 10)
(setf X (list X))
```

Such a loop provides X with a potentially unbounded set of values (as well as types!)

This difficulty with iteration has been addressed in several ways. First, many kinds of loops can be represented by a closed form expression which accurately characterizes the effect of the loop on the symbolic values of variables. These expressions are computed by the solving of a set of recurrence relations [CR81].

Many other kinds of loops are not amenable to such analysis, however. An alternative solution is to place an arbitrary limit on the number of times a loop is symbolically executed. Such an approach handles broader classes of loops than those whose recurrence relations can be solved, but at the cost of accuracy: only the approximate effects of the loop on variable values is represented.

A third kind of solution involves the use of "well-founded property sets" [Weg75]. In this approach, it is the symbolic values which are approximated, rather than the number of times through the loop. The values are organized into a lattice, and each time through the loop, the symbolic values of variables either stabilize or move upward in the lattice. The finite lattice structure guarantees that the symbolic values of variables will eventually reach a fixed point after some number of iterations. The sacrifice made by this technique, of course, is that range of variable values represented by the system is highly

constrained. The "typestates" mechanism developed for the NIL language [Str83] follows this same general approach to approximating the actual type of an object.

Once some way of representing the effects of conditionals and iteration has been chosen, the representation of the program produced is then analyzed to detect *anomalies*—situations, such as an undefined variable reference, that indicate some form of error in the program. The DAVE system [OF76] exemplifies this work. It analyzes FORTRAN programs to detect occurrences of the anomalous sequences of events, such as uninitialized variables, or redundant definitions (such as two definitions without an intervening use, or a definition without any subsequent use). This work has been generalized in CECIL [OO90], which provides a regular expression-based language for expressing user-specified sequencing relationships, rather than the hard-wired relationships uncovered by DAVE. This work relates to other sequence representation work, such as constrained expressions [ADWR86].

Type flow analysis vs. value-based symbolic execution

The obvious difference between type flow analysis and value-based symbolic execution is its concern with representation of the types of variables, rather than their values. This has an immediate benefit: the problems with loops are somewhat ameliorated, since loops are much more likely to have fixed points when interpreted at the type level than when interpreted at the value-level. For this reason, the current implementation of type flow analysis choses the fixed number of iterations approach to resolving the iteration conundrum. Another result of type, rather than value-level, analysis is the

use of a parametric type language in conjunction with a type predicate-based path condition language, instead of the algebraic expressions used in value-level processing. Finally, the need for a sophisticated sequencing language is largely absent in type flow analysis for a very simple reason: type-level errors are order-independent, unlike the errors pursued with value-based symbolic execution systems like DAVE or CECIL. In DAVE, for example, the distinction must be made between a path where a variable is referenced followed by its definition, and a path where a variable is defined followed by its reference: the first case indicates an error, while the second is normal and correct. In type flow analysis, it makes no difference whether a variable is first used as an integer and subsequently used as a string, or first used as a string and then used as an integer: an error is indicated regardless.

Though the domain of type flow analysis simplifies some of the problems that occur with value-level processing, it also introduces new ones. For example, though algebraic manipulation is eliminated, the manipulation of polymorphic type expressions is introduced, with the need for unification-based constraint maintenance. Second, while DAVE and CECIL represent control and data flow for the monomorphic, non-higher order language FORTRAN, type flow analysis is applied to the polymorphic, higher-order language Lisp. This requires the representation of functions as data, for example. These extensions will be examined in more detail in subsequent chapters.

2.3.2 Type-level symbolic execution

The concepts of value-based symbolic execution have also been applied at the type-level. An early example of this work is [Ten74]. The primary

application of this work is to disambiguate overloaded operators in the SETL programming language. In SETL, the operator + not only represents numeric addition, but also concatenation for strings and union for sets. A significant source of run-time overhead in SETL results from the inspection of operands to determine which implementation of + to invoke.

To accomplish the disambiguation, Tennenbaum defined a simple type lattice with a distinguished null type at the bottom, a set of atomic types like integer, string, character, and set, and a distinguished "most general type". The nodes between the atomic types and the most general type represented disjunctions of atomic types, and compositions of the set type with the atomic types. In a manner similar to well-founded property sets [Weg75], the analysis of the SETL program produced a bound on the kinds of types of values that a given variable could take on.

This bound was often rather approximate. For example, since the program represented the disjunction of two types as their join in the lattice, a variable that could take values of type *set of integer* or *set of string* would be represented by the most general type—in other words, the program could not constrain the type of values on that variable at all. More importantly, the program was a type *finder*, not a type *checker*—it assumed that the input program was type consistent to begin with. This work was extended and formalized in [KU78].

Later work addressed some of the issues involved in recognizing type level errors. [Mil78a] recognized that the forward and backward directions of information flow could be used to generate qualitatively different kinds of type information. Consider the assignment statement `(setf X Y)`. If `X` is known through prior analysis to be an integer, then information flow in the same

direction as execution propagates that type to Y. Forward information flow is the strategy classically adopted in value-based symbolic execution.

Backward information flow, on the other hand, produces constraints on what the types must be, rather than direct information about what they are. If, in the assignment (setf X Y), it is known through prior analysis that Y is an integer, then information flow in the reverse of execution paths would propogate that type to X.

Miller's technique thus involves generating, for each node in the control flow representation, both the forward information flow-derived information and the backward information flow-derived information. Each of these is examined both separately and together. Type inconsistency can be inferred when the information accumulated through forward inference is inconsistent, or when the intersection of the forward and backward derived type information for a variable at some node is inconsistent.

In addition, these two types of information can also be used to determine whether run-time type checks are required. This occurs when the information about the types gathered through forward information flow is insufficient to guarantee the type requirements procured through backward information flow.

Exploiting both forward and backward information flow is quite important in type flow analysis: declaration-based and type predicate information is used in forward information flow, while constraint-based information is derived from backward information flow. In common with the value-based symbolic execution work, however, Miller applied his techniques to a monomorphic, non-higher order language, and so did not address the issues of polymorphism and functional objects confronted by type flow analysis.

More recently, Beer has applied this distinction between forward and backward inference to the compile-time type analysis of Common Lisp programs [Bee88]. This work, while sharing the same target langauge as type flow analysis, as well as a concern with types, still differs markedly in goals and scope. Like Tennenbaum, Beer is concerned with the run-time overhead incurred by overloaded operators—here the numeric operators that can operate on a variety of number representations: fixed-width integers, variable width integers, floating point numbers, etc. Beer's research involves the analysis of Lisp programs with the goal of inferring the optimal representation for the numeric values manipulated by variables in the program. In certain cases, such derived information produces compiler optimizations that halve the execution time of the unanalyzed program.

Finally, Narayanaswamy has done some work on exploiting the control flow relationships between Lisp functions to support exploratory software development in the Common Lisp Framework environment [Nar88]. The scope of the aid provided by this system is severely circumscribed by the fact that it represents *only* control flow relationships, without data flow. Such information helps to implement support such as queuing functions to edit when parameter lists are changed, but falls short of any real structural assessment capabilities.

Type flow analysis vs. type level symbolic execution

Type flow analysis shares with type level symbolic execution a concern for both forward and backward information flow. However, type flow analysis differs in the kinds of type level information being represented, and the usage made of this information. The work of Tennenbaum and Miller is primarily

concerned with the optimization of the code generated by the compiler for a monomorphic, non-higher order language. Similarly, Beer's research is limited to the monomorphic, non-higher order data objects in Common Lisp.

Because of this optimization emphasis, there is no need to represent data types at a higher level of abstraction than the predefined primitive types and operations. From the standpoint of a compiler optimizer, it doesn't matter if the array being manipulated is "really" a stack from the standpoint of the user. Thus, the information hiding provided by user-definable types is not preserved in the analysis systems.

Finally, these techniques provide little help for assessing the structural correctness of the program: Tennenbaum and Beer essentially assume that the program to be analyzed is structurally correct, while Miller's technique is applied to a monomorphic programming language without facilities for data abstraction. Of course, assessing structural correctness falls outside the purview of a compiler optimization technique.

This review of "type-level symbolic execution" research has neglected polymorphic type inference, a large body of work employing flow analysis techniques for assessing the type consistency of polymorphic, higher-order programming languages. Part of the reason for this is to structure the review along historical boundaries: polymorphic type inference was developed within the functional language community, while symbolic execution systems arose to address the needs of imperative languages. This difference in domain led not to just different goals but different techniques as well. Finally, while issues in polymorphism and functional data objects suffice to distinguish type flow analysis from the "imperative" symbolic execution systems above, differ-

ent and finer distinctions must be drawn between type flow analysis and the functional, polymorphic type inference systems to be described next.

2.4 Type flow analysis as type inference

Functional languages, such as ML [GMW79], Hope [BMS80], Miranda [Tur85], and the recently designed and designated "standard" Haskell [HWA+90], are principally distinguished from imperative languages like Pascal or Ada by not including assignment statements or mutable data. Such omissions lead to the property of *referential transparency*, which roughly means that the result produced by an expression is independent of the context in which it appears. This property, in turn, makes functional languages more formally tractable than imperative languages. Perhaps the biggest success of this amenability to formal treatment is the use of type inference, rather than programmer-supplied type declarations, to verify the type consistency of programs. Not only is type information inferred rather than declared, but the inference scheme supports *polymorphism* to a degree unobtained by strongly typed imperative languages.

Polymorphism means to have many forms. In functional languages, this takes the form of allowing functions to accept and return a range of types. The two classic flavors of polymorphism are termed *parametric* and *ad hoc* [Str67]. Parametric polymorphism refers to a function that operates over a range of types in a uniform manner—in concrete terms, the same code is executed independent of the type of value passed. The `length` function is the

archtypal parametric polymorphic function, which executes uniformly over lists of integers and lists of floating point numbers.

In contrast, ad hoc polymorphic functions accept a range of possibly unrelated types, potentially dispatching to different code for each kind of type passed. Primitive numeric operations like + and * are commonly ad hoc polymorphic, since they accept both integer and floating point numbers and execute different code depending upon the type passed. [CW86] contains more details on the several forms of polymorphism.

The seminal paper on polymorphic type inference is [Mil78b], which presented the algorithm utilized in the proof system LCF that spawned the language ML. ML, like the other functional languages of its generation (such as Hope and Miranda, but excluding Haskell), can infer the type of parametrically polymorphic functions, but not ad hoc polymorphic functions. (The type of traditionally ad hoc polymorphic functions, such as numeric operations, is hard-wired into the language processor.)

This "basic" type inference mechanism rests on the use of a relatively simple type expression language consisting of type variables (commonly denoted by α, β, etc.), atomic types (such as integer and string), and parametric types (such as list[α] and list[integer]). The inference process can be viewed as providing each identifier in a function with a unique type variable, and then letting these variables accumulate constraints through the contexts in which their associated identifiers occur. Unification [Rob65] is used to both accumulate the sets of constraints on the type of an indentifier through its contexts of use, and to ensure that these separate usages are consistent. From this standpoint, type inference consists of the process of solving sets of simultaneous equa-

tions representing the type constraints to be satisfied by the expressions and subexpressions in the program. Several good descriptions of the parametric polymorphic type inference algorithm have been published: [Han87] describes a functional implementation of the algorithm in Miranda, while [Car87] constructs an imperative implementation in Modula-2.

Three important properties of the parametric polymorphic type inference algorithm are semantic soundness, syntactic soundness, and completeness. They rest upon the notion of a *most general* typing, also known as the *principal* typing [DM82]. In a polymorphic language, it is important that the type inference system find the most general type possible for an expression. For example, the identity function has both type $\alpha \rightarrow \alpha$ and integer→integer, though only the first typing is the most general typing. "Well typed" expressions are those for which a most general typing exists.

Syntactic soundness implies that if the algorithm finds a typing for an expression, the typing is the most general typing. Semantic soundness implies that well typed expressions are free from type errors. Finally, completeness implies that if an expression has a well typing, the algorithm will find a typing that is at least as general. Informal proofs of these properties are presented in [FH88]. Together they imply that parametric polymorphic type inference will accurately type all legal programs in the target functional language, and that failure to type a program proves the existence of a type error in the program.

Parametric polymorphic type inference vs. type flow analysis

In general, type flow analysis adopts the type expression language of parametric polymorphic type inference, while extending the control and data flow

machinery to accomodate the capabilities for ad hoc as well as parametric polymorphism in exploratory languages like Lisp. These extensions result from the need to represent multiple execution paths at the type level, and the effect of type predicate constructs in the language.

The representation of the type of a function in parametric polymorphism is simply a labelled tuple, where a single type expression is associated with each parameter to the function, in addition to an expression representing the return type. For example, the identity function has the type $\alpha \rightarrow \alpha$ and the standard mapping function has the type:

$$\alpha \rightarrow \beta \rightarrow \mathsf{list}[\alpha] \rightarrow \mathsf{list}[\beta]$$

Ignoring the implicit representation of currying[3], this type expression indicates that map takes a function from α to β, and a list of α, and returns a list of β.

The crucial implication of this representation is that *parametric polymorphic type inference supports only functions containing a single execution path at the type level.* This is manifest in the type inference mechanism, which ensures that each branch of a conditional is constrained to return the same type, and that the result of loops is invariant over the number of iterations at the type level. It also manifests itself in the language definition, which excludes type predicates, since their presence enables the ability to "do something different" based upon the type of object passed. Parametric polymorphism is

[3]Currying is a powerful feature of some functional languages, where the invocation of an n-ary function with less than n arguments returns a new function which "closes over" the supplied arguments. For example, if + is a binary operation, then the expression (+ 1) represents a unary function that increments its argument by one.

defined on the idea that a function can accept a range of types only when the specific type passed is irrelevant to the computation being performed.

Type flow analysis recognizes that for languages like Lisp, the representation of functional type must take into account the possibility of multiple execution paths at the type level. Furthermore, which execution paths are possible in any particular context is mediated by the occurrence of type predicates, which perform a filtering on the objects passing the conditional test form and subsequently executing the conditional body. Thus, instead of representing functional type as a single tuple of type expressions, type flow analysis employs a more sophisticated representation involving a set of execution path structures, where each execution path contains not only a set of type expressions corresponding to the parameters to the function, but a path condition as well. Standard parametric polymorphic functions, such as the `length` function, are represented by type flow analysis as a function containing only a single execution path and without any path conditions (Figure 6).

This increased level of generality has a cost: since types can be manipulated in a similar manner to values in Lisp, precisely representing the type-level execution of Lisp programs requires precisely representing the value-level execution, as will be discussed in Chapter 5. Since the value-level representations of symbolic execution systems are inherently imprecise, this ambiguity propagates to type flow analysis. The result is that type flow analysis cannot hope to enjoy the formal properties of syntactic soundness, semantic soundness and completeness without restricting the expressive scope of the language.

- *The standard parametrically polymorphic length function:*

```
(defun length (lst)
  (if (endp lst)
      0
      (+ 1 (length (cdr lst)))))
```

- *The parametric polymorphic representation:*

```
LIST[TV-1] -> INTEGER
```

- *Type flow analysis represents parametric polymorphism as a braid containing a single execution path with no feasibility conditions:*

$$Br_1 \left[St_{11} \left\{ \begin{array}{ll} FC: & T \\ CE: & lst : \mathsf{list}[\mathsf{v}_1] \\ RT: & \mathsf{integer} \end{array} \right\} \right]$$

Figure 6: The parametric and type flow representations of LENGTH

Type classes and other variations on parametric polymorphic type inference

Extending the basic parametric polymorphic inference paradigm is currently a major focus of research in the programming language community. For example, [Mac86] extends the mechanisms for expressing type level structure to support module-level structure of programs. Much research centers on the extension of the type system to support various forms of inheritance. [Gra89] investigates its extension to accomodate Smalltalk inheritance, while [XW88] and [BTCGS89] develop techniques for logic programming languages.

The most relevent extension to parametric polymorphic type inference is called *type classes*, a mechanism first proposed in [WB89] and incorporated into the language definition of Haskell [HWA+90]. Type classes are a solution to the problem of ad hoc polymorphism of the kind exhibited by common numeric operations, and normally resolved in, well, an ad hoc manner. Type classes allow the definition of overloaded operators where the resolution of the overloading can be accomplished statically and in keeping with the techniques of parametric polymorphic type inference.

For example[4], suppose the numeric operation * should support both floating point multiplication (`float-*`) and integer multiplication (`integer-*`). As shown in Figure 7, this can be accomplished by defining a type class, `Num`, which supports the operation *. The type class forms a kind of template, where the variable a is substituted with a type expression at compile-time. The range of legal substitutions is indicated through `instance` declarations,

[4]This example adapted from [WB89].

• *The class declaration for the numeric operation *:*

```
class Num a where
  (*) :: a -> a -> a

instance Num Int where
  (*) = integer-*

instance Num Float where
  (*) = float-*
```

Figure 7: Example type class declaration

which declare the bindings of the type class variables. As the figure shows, two legal overloadings for * are declared through two `instance` declarations.

Now consider the use of * in a function such as `square`:

```
square x = x * x
```

This function is given the following type:

```
square :: Num a => a -> a
```

which means "`square` has type `a -> a`, for every `a` such that `a` belongs to class `Num`." The Haskell expression `square 3` will have type `integer`, while `square 3.14` will have type `float`.

An important property of type classes is that every program employing them can be translated at compile-time into an equivalent, non-type class program. This equivalent program can be typed using standard parametric

polymorphic type inference, and thus enjoys the formal properties of that technique: syntactic soundness, semantic soundness, and completeness.

Type classes form a significant extension to basic parametric polymorphic type inference, and constitute an acknowledgement that ad hoc polymorphism is a useful form of genericity. However, when compared to type flow analysis, they support only a very specialized form of ad hoc polymorphism. This specialization consists of a type-based dispatching conditional, where each test clause consists of a single type predicate test on each argument. For example, the Num class and associated instances result in behavior similar to the following Lisp definition of *:

```
(defun * (x y)
  (cond ((and (integerp x) (integerp y))
         (integer-* x y))
        ((and (floatp x) (floatp y))
         (float-* x y))))
```

When viewed from this perspective, it becomes clear that type flow analysis removes two important restrictions from the kinds of ad hoc polymorphism supported by type classes.

First, type flow analysis allows arbitrary kinds of conditional test clauses. This means that or and not are also allowed as logical connectives. In addition, the arguments to the type predicates need not be the parameters to the function, but may also be arbitrary expressions and/or references to global values. Finally, both type-based computation (i.e. type predicates) as well as value-based computation (i.e. non-type predicates) can be mixed in arbitrary ways in a conditional test clause.

Second, type flow analysis supports the ability of a given set of argument

constraints to succeed against more than one conditional test. Type classes, in keeping with their compile-time translation of the program to a parametrically polymorphic equivalent, require complete disambiguation. In some sense, all occurrences of * must be "macro-expanded" at compile time into a call to exactly one of `integer-*` or `float-*`. While type flow analysis will have this same effect for these kinds of functions, it also supports programs where such disambiguation occurs only at run-time. For example, consider a different version of * which accepts two integers and returns a float if the result is larger than the largest integer. Such "value-based" ad hoc polymorphism cannot be represented within type classes, but is easily represented with type flow analysis.

2.5 Summary

Type flow analysis is an attempt to improve the exploratory software development paradigm by providing an incremental, evolutionary approach to improving the robustness of exploratory programs. In so doing, it adopts the language and development style of AI programming, and imports an analysis capability that blends polymorphic type inference and symbolic execution. Type flow analysis for exploratory software development thus constitutes a novel research thrust within the subdiscipline of AI and SE.

Type flow analysis employs some representations and techniques common to symbolic execution systems. Principal among these is the representation of the set of execution paths in the software system, and the use of path conditions to distinguish between feasible and infeasible execution paths. Unlike most

symbolic execution systems, type flow analysis concerns itself with type-level representation of control and data flow, with the representation of polymorphic functions, and with functions as first-class data objects.

Type flow analysis also bears similarities to polymorphic type inference. The type expression language for non-functional data objects is similar to that developed for parametrically polymorphic languages like ML. Unification is used to compare instances of data objects, to gather constraints, and to assess consistency. However, polymorphic type inference restricts functions to a single execution path at the type level, and allows only parametric polymorphism. Extensions of polymorphic type inference to support overloaded operators still require compile-time resolution to a parametric polymorphic equivalent. Type flow analysis eschews these restrictions in supporting Lisp, a language with capabilities for a much broader range of ad hoc polymorphism. The trade-off is in the tightness of the analysis: polymorphic type inference has been formally verified to be syntactically sound, semantically sound, and complete. These properties mean that all legal programs can be typed, and failure to type the program proves a type error. Since types in Lisp can be manipulated in a similar manner to values, such formal properties cannot be achieved for type flow analysis.

CHAPTER 3

THE MECHANISMS OF TYPE FLOW ANALYSIS

This chapter begins the detailed presentation of type flow analysis for exploratory software development. This chapter is concerned with describing the fundamental representations and mechanisms in ESSIE, the current implementation. These representations and mechanisms consist principally of: the type expression representation language; the braid and execution strand representation language; and the mechanisms that manipulate these representations and implement the three basic programming language constructs: sequencing, conditionals, and iteration.

This chapter abstracts away the details of actual data and control flow constructs in a real programming language. The next chapter on the language of type flow analysis presents concrete language constructs that allow a type analysis system implementor to design specific control and data flow mechanisms. The subsequent chapter illustrates how a particular set of control and data flow definitions were employed in the analysis of an example software system.

The next two sections describe the representational machinery for data objects, which are modelled by a *type expression language*, and control flow, which is modelled by sets of *execution strands*. These sections are followed by

an introduction to type flow processing, which is in turn followed by detailed looks at the processing of the three fundamental language constructs in type flow analysis.

3.1 The type expression language for data object representation

ESSIE defines a type expression language for the representation of the types of data objects. This type expression language is essentially equivalent to the parametric type expression language used in languages like ML. Other descriptions of parametric type expression languages are found in [Car87] and [Han87].

Type expressions are built from three basic kinds of objects: type variables, primitive types, and constructed types. Type variables stand for an arbitrary type expression, and are denoted by v_1, v_2,...,v_n. Primitive types are types without internal type variables, such as integer and string. Constructed types support the composition of types, such as list[v_1], or list[list[integer]][1].

3.1.1 Contrasts with parametric type expression languages.

While this type expression language is similar to those of parametric polymorphic inference systems, its embedding within the control flow representation machinery of type flow analysis leads to significant differences.

[1]Type expressions are printed in sans serif font, while program text is printed in `typewriter` font.

First, parametric polymorphic inference systems associate each identifier in a function definition with a single type expression—in other words, a single type expression must suffice to represent the identifier's type over *every occurrence* of the identifier in the function definition. In contrast, ESSIE associates a single type expression with each identifer *within a particular execution path.* This means that different occurrences of the same identifier within a function may be associated with completely different types, as long as these occurrences fall along distinct execution paths.

Second, parametric polymorphic type expression languages represent only *homogeneous* data objects through constructed types. For example, they can represent lists of integers or lists of strings, but not lists that contain both integers and strings. While ESSIE's current type expression language cannot directly represent such *heterogeneous* data objects either, it synergistically combines with ESSIE's control flow representation to allow such objects to be approximately represented. For example, a list of integers and strings can be represented as a list containing an unordered set of integers and strings, where any manipulation of the interior elements of the list must support the manipulation of either integers or strings. The capabilities and limitations of ESSIE's support for heterogeneous data objects will be discussed further in the next two chapters.

3.1.2 Inferring type expressions through contextual, definitional, and type predicate-based information

The goal of any type inference system is to associate type expressions with the identifiers in the system. In ESSIE, this type information is inferred

from three sources: *contextual* information, *definitional* information, and *type predicate* information.

Contextual type information is gathered from the environment surrounding the identifier under study—for example, in the form `(car x)`, `x` can be inferred to be an object of type list, since `car` requires arguments of that type. Contextually-derived information is also termed *operator-based*, since it arises from the propagation of type constraints on the parameters of data operators to their arguments. The mechanisms for correctly propagating this information will be discussed in this chapter.

Definitional type information is produced from the functions that create data or functional objects. For example, the result type of the form `(cons 'a '(b))` is list[symbol], which is inferred from the definition of the `cons` function. Definitional information can propagated in various ways— the result type of the following form is inferred to be of type list[integer] through definitional means:

```
(let ((x (list 10)))
  x)
```

Definitional type information is also termed *constructor-based*, since it results from the analysis of a constructor of data, such as `list`, or a constructor of functions, such as `defun`.

The third way of inferring type expressions is through type predicate-based information. The occurrence of type predicates in conditional clauses provides information about the type expressions that must be associated with identifiers along the execution path following the conditional test. Consider the following conditional form:

```
(cond ((integerp x)
       (miscellaneous-operation x)))
```

In the body of the conditional arm, x must be an integer, since for that body to be executed, the form (integerp x) must have succeeded. This form of information differs completely from contextual mechanisms: no type expression information is gathered on x through constraints on the argument of integerp, since this function accepts arguments of any type. Type predicate-based information also differs from definitional information, since the result type returned from integerp is not bound to its argument. Finally, the actual type expression information obtained from type predicates is closely entwined with the structure of the conditional test clause itself. This will be thoroughly discussed in Section 3.5 on conditional type flow processing.

3.1.3 Unification

In ESSIE, *unification* is used both to accumulate constraints on the types associated with identifiers, and to determine structural inconsistencies. In general, unification is a pattern matching technique that answers the question, "How can two expressions be made equal?" This equality is achieved by finding a set of substitutions for the variables occurring in the two expressions. For example, given the expressions list[integer] and list[v_1], substituting integer for v_1 in the second expression yields equality of the two expressions. *Instantiation* is the process of actually carrying out the substitutions found by unification. Continuing the example above, the expression list[v_1] would become list[integer] by instantiating v_1 to integer.

ESSIE uses unification and instantiation to incrementally build the type expression associated with each identifier. For example, ESSIE begins processing function definitions by initializing each parameter to a distinct type variable within a single execution strand, and then using definitional, contextual, and type predicate-based information to unify and instantiate this type variable to other type expressions.

Failure of unification indicates a structural error in the source code. For example, consider the following expression:

```
(+ 1 x (first x))
```

This expression is structurally inconsistent because, within a single execution path, it requires the type of x to be simultaneously an integer (since x is an argument to +), and a list[v_n] (since x is an argument to `first`). ESSIE's type constraint language and unification uncovers this structural error in the following general way.

Assume that x was previously unconstrained and bound to the type variable v_2. The first step involves unifying v_2 with the type expression assigned to the parameter to `first`, list[v_3]. This unification leads to the instantiation of v_2 to list[v_3], which effectively updates x to this latter expression. X also appears as an argument to the function +, which imposes the type constraint integer on all of its arguments. This leads to the attempted unification of integer with list[v_3], which fails, since there is no way to instantiate v_3 to something that makes the two type expressions equal.

[Kni89] is an excellent survey article on unification. The next section discusses the basic representations for control flow in ESSIE, and how this type expression language is contained within it.

3.2 Execution strands and braids for control flow representation

The inclusion of type predicates in exploratory languages allows the definition of functions that inspect the types of data objects at run-time and perform different actions depending upon the results. This requires a very different representation for their type-level structure than that used for languages such as ML. In these latter languages, functions can be represented in the same manner as compound data types such as lists.

ESSIE represents the structure of a function as a *braid*—an object that represents the different type-level execution paths through the function. Each braid consists of a set of *execution strands*, where each strand has three parts—a feasibility condition, an environment of constraints, and a result type:

- **Feasibility condition.** This object represents a predicate that determines whether this execution strand represents an execution path that could, in principle, be executed. The feasibility condition consists of a set of *predicate-expression pairs*. The first element of a predicate expression pair is a type predicate such as `listp`, `integerp`, etc. The second element is a type expression originally computed from the argument to the type predicate. Note that the introductory chapter simplified the representation of feasibility conditions for expository purposes. The previous representation, which paired type predicates with parameter names, could not represent situations in which the argument to a type predicate was a complex, possibly polymorphic expression such as `(integerp (push item stack))`.

- **Constraint environment.** This object provides bindings between the local variables to a function (i.e., the parameters, and any let-bound identifiers) and their type constraint expressions. The constraint environment also holds the local contextual constraints gathered from recursive calls to the function itself. The constraint environment only holds local information—the structure of globally defined functions is stored in another, globally accessible structure. Finally, the constraint environment is organized as a stack of *binding frames* to model the standard scoping rules and identifier shadowing capabilities of `let` forms.

- **Result type.** This object represents the type of object that would be returned from the execution strand. It consists of a single type expression.[2]

3.2.1 Notational conventions for braids and strands

In the remainder of this chapter and the next, braids, execution strands, and strand components will be manipulated in many different ways. The best representation for these structures depends upon the particular mechanism under discussion. To provide clarity and perspicuity, the following multi-level notation system will be used.

The description of some mechanisms needs only distinguish between different instances of a structure, without reference to their internal components. For braids, such *stand-alone* references are denoted Br_1, Br_2,... Br_n. Stand-alone references to execution strands are denoted by St_{ij}, meaning the jth

[2]ESSIE does not currently represent the ability in Common Lisp to return *multiple values* from a function. Such an extension would be supported by allowing the result type component of execution strands to contain a set of type expressions.

execution strand in Br_i. The stand-alone references to execution strand components are similarly subscripted: FC_{ij}, CE_{ij}, and RT_{ij} refer respectively to the feasibility condition, constraint environment, and result type for the jth execution strand of Br_i.

One way of representing the internal structure of a braid is to simply identify the component strands:

$$Br_i[St_1\ St_2\ \ldots\ St_n]$$

The "[" and "]" delimit the internal structure of the braid. The stand-alone form of these same execution strands would be $St_{i1}\ldots St_{in}$.

The representation of the internal structure of an execution strand St_i can take two forms. One form symbolically refers to its components:

$$St_i\{FC_i\ CE_i\ RT_i\}$$

In contrast to braids, the internal structure of strands is delimited by "{" and "}". Relatively simple strands are represented on a single line as follows:

$$St_1\{\ \mathrm{T}\ \langle\ \mathsf{obj} : \mathsf{integer}, \mathsf{item} : \mathsf{v}_4\ \rangle\ \mathsf{string}\}$$

In this form the delimiter "⟨" separates the path feasibility condition from the constraint environment, and the delimiter "⟩" separates the constraint environment from the result type.

More complicated strands require multiple lines, and so the feasibility condition, constraint environment, and result type are explicitly labelled FC, CE, and RT, respectively:

$$St_{ij}\left\{\begin{array}{ll} FC: & (listp\ \mathsf{list}[\mathsf{v}_1]) \\ CE: & item : \mathsf{v}_1; stack : \mathsf{list}[\mathsf{v}_1] \\ RT: & \mathsf{list}[\mathsf{v}_1] \end{array}\right\}$$

Finally, an example of a fully expanded braid is the following:

$$Br_i\left[\begin{array}{l} St_1\{\ \mathrm{T}\ \langle\ \mathsf{obj} : \mathsf{integer}, \mathsf{item} : \mathsf{v}_4\ \rangle\ \mathsf{string}\} \\ St_2\{\ \mathrm{T}\ \langle\ \mathsf{obj} : \mathsf{string}, \mathsf{item} : \mathsf{v}_6\ \rangle\ \mathsf{integer}\} \\ St_3\{\ \mathrm{T}\ \langle\ \mathsf{obj} : \mathsf{string}, \mathsf{item} : \mathsf{v}_9\ \rangle\ \mathsf{string}\} \end{array}\right]$$

Producing a braid representing the structure of a function involves processing sequential, conditional, and iterative language constructs. The next section presents a simple example of the analysis process at a high level. Following this section are more detailed descriptions of the analysis mechanisms for type flow application, conditional clause processing, and recursion processing.

3.3 An overview of the mechanisms of type flow analysis

Figure 8 illustrates how the three parts of an execution strand accumulate information as the function `double-reverse` is processed. In this example, it is assumed all of the functions called by `double-reverse` have been analyzed previously, and their braid structure is available for use during the processing of this function. The braids for these functions are provided at the top of the figure.

Pre-analyzed function braids

```
<: [ { T <param1 : integer ; param2 : integer > symbol }]
*: [ { T <param1 : integer ; param2 : integer > integer } ]
-: [ { T <param1 : integer ; param2 : integer > integer } ]
string-append : [ { T <param1 : string ; param2 : string > string } ]
reverse : [{ (integerp integer) <param1 : integer > integer }
            { (stringp string) (not (integerp string)) <param1 : string> string }]
```

```
(defun double-reverse ( x )
                                        [ { T <x : v1 > v2 } ]                          1
  (let ((doubled-x  (cond ((integerp x)
                                        [ { (integerp v1) <x : v1 > symbol } ]          2
                          (cond ((< x 0)
                                        [ { (integerp integer) <x : integer > symbol } ]3
                                 (- (* 2 x)))
                                        [ { (integerp integer) <x : integer > integer } ]4
                                (t
                                        [ { (integerp integer) <x : integer > symbol } ] 5
                                 (* 2 x))))
                                        [ { (integerp integer) <x : integer > integer } ]6
                        ((stringp x)
                                [ { (not (integerp v1)) (stringp v1) <x : v1 > symbol } ]7
                         (string-append x x)))))
                       [ { (not (integerp string)) (stringp string) <x : string > string } ]8


    [ { (integerp integer) <doubled-x : integer ; x: integer> integer }                 9
      { (not (integerp string)) (stringp string) < doubled-x : string ; x : string > string } ]
   (reverse doubled-x)

    [ { (integerp integer) <doubled-x : integer ; x : integer > integer }               10
      { (not (integerp string)) (stringp string) < doubled-x : string ; x : string > string } ]
  )

  [ { (integerp integer) <x : integer > integer }                                       11
    { (not (integerp string)) (stringp string) <x : string > string } ]
)
```

Figure 8: Execution strand processing in double-reverse

The braids for the arithmetic operations `*`, `minus`, and `<` each consist of a single execution strand, each of which in turn has a `T` path predicate. This indicates that these functions do not perform any checks upon the type of the arguments through type predicates. However, all of these functions require their arguments to be integers[3], as represented in the constraint environment. Finally, the `*` and `minus` functions return integers, as indicated in their result values, while the `<` function returns a symbol (whose value is `t` or `nil`—as in Common Lisp[4]).

The braid for the function `reverse` (with a single parameter `param1`) is somewhat more complicated: it consists of two strands, the first of which represents the execution path taken when an integer argument is "reversed", and the second of which represents the execution path taken when a string is reversed. The two feasibility conditions associated with these strands operate as "guards" on the strands, ensuring that integer arguments do not proceed down the strand for strings, and vice-versa. The mechanisms for feasibility condition manipulation will be discussed in detail below. Finally, the result types for the two strands represent the fact that `reverse` returns a string when passed a string, and an integer when passed an integer.

Below the listing of the structure of the pre-analyzed functions is the definition of the function `double-reverse`, annotated with some of the structural information accumulated during the processing of the function. The following describes how each of these braids (as numbered from 1 to 11 in the figure) is formed:

---

[3]A simplification for expository purposes. Chapter 4 will discuss the language constructs provided to represent such ad hoc polymorphic functions more accurately.

[4]Some Lisp dialects do have a boolean type—Scheme, for example.

(1) This first, "initialization" braid represents the information accumulated by processing the function header, which consists of declaring the identifier x. No constraints on x or on the result type of the computation exist yet.

(2) The second braid represents the result of processing the first conditional's test clause. To produce this braid, execution braid 1 is combined with the information from the test clause. The resulting addition to the feasibility condition indicates that a pre-condition on the validity of this execution braid is that x be an integer. The result type of symbol represents what would be returned (i.e., t or nil) if the conditional arm had no body.

(3) This braid is built from execution braid 2 and the processing of the first conditional test form of the interior cond clause. This braid is interesting because the processing of the test clause changes the constraint environment.

(4) This braid results from the processing of the minus form in the body of the interior cond's first clause, based upon execution braid 3. X is again constrained to an integer in this form, and the result type is changed, since minus returns an integer.

(5) This braid processes the t clause of the interior cond form, and thus is built from execution braid 2, the one immediately preceding the interior conditional. The feasibility condition for the single strand in this braid is built from the feasibility condition for braid 2 and any type-level filtering performed by the current condition test. In this case, no type filtering is accomplished by the t clause (or by the test clause preceding it), so braid 5 simply maintains the feasibility condition from braid 2. Its constraint

environment carries information from the test clause (< x 0) for the same reason that braid 3 does. Section 3.5 describes this processing in more detail.

(6) This braid follows from the processing of the body of the t clause for the interior conditional, and is built from execution braid 5. The multiplication operation redundantly constrains the argument x to an integer (it was previously constrained to an integer by the form (< x 0)), and newly constrains the return type of the strand to an integer.

(7) This braid combines execution braid 1 with the conditional clause test stringp. The resulting execution braid's feasibility condition clearly shows how, in order for execution to reach the stringp test, it must fail the integerp test. The result type of symbol reflects the type of value (i.e., t or nil) returned by the conditional test clause.

(8) This braid is built from braid 7 and the string-append form, and constrains the type of x to a string, as well as providing a string result type.

(9) Up to this point, the processing of each form was built upon an execution braid with a single execution strand, since there was only one execution path to each form. Execution of the reverse form, however, could be arrived at from any of three points, corresponding to the 3 arms of the nested conditionals. These three braids (4, 6, and 8) are collected together here. Note that since the execution strands in braids 4 and 6 are structurally identical, only one of them shows up in the combined braid 9. Also, the constraint environment now contains a binding for the identifier doubled-x.

(10) This braid represents the result of processing the `reverse` form. Note that it represents the composition of the two-stranded braid in **9** with the two-stranded braid for `reverse`. Of the four potential composite execution paths, only two are feasible, which are shown in braid **10**.

(11) This braid results from exiting the `let` form, which has the effect of "popping" the binding frame containing the binding for the identifier `doubled-x` from the constraint environments in the execution strands. It also represents the braid to be stored as the type-level structure of `double-reversed`.

This description of the analysis of the `double-reverse` function provides the general idea of type flow analysis: an initial environment is set up and the statements in the function are sequentially processed. Conditionals result in the splitting of type flow, and `let` forms define new lexical environments. Mechanisms for the elimination of redundant and infeasible execution paths are present. However, this example omits many minor details, such as how conditional test forms and function applications are *really* processed, as well as a major detail: iteration through recursion. The following three sections present a detailed look at sequential, conditional, and iterative processing, filling in these gaps.

3.4 Sequential type flow processing

This section describes how the type flow of a sequence of forms is processed in ESSIE. Sequential processing is accomplished rather simply, assuming a mechanism for *type flow application*, which takes a form and a braid repre-

senting the set of execution paths up to that form, and then returns a new braid representing the execution paths through the form. This section begins by discussing type flow application in more detail.

Function application refers to the invocation of a function on a set of arguments during program execution to compute a value, which is returned. Type flow application is the analysis-level counterpart to function application: it refers to the type-level "invocation" of a function on its arguments, returning a new type-level structure as a result.

Type flow application propagates type constraints in two directions. First, constraints are propagated from parameters to their corresponding arguments. For example, if an argument is bound to the type expression v_1, and the corresponding parameter to the function is bound to integer, then type flow application may lead to the instantiation of v_1 to integer.

Another direction of type level information flow is from arguments and parameters to return value. For example, if the function `first` is invoked with a value whose type expression is list[integer], then the result type of the function invocation will be integer.

This information propagation is common to both polymorphic type inference and type flow application. However, while polymorphic type inference requires a one-to-one correspondence between arguments and result type expressions, type flow application supports one-to-many. In other words, applying a function to its arguments at the type level may result in a *range* of possible result types. This accommodates functions such as push in Chapter 1, where passing a stack represented as a list could result in the return of either a list or a vector.

In fact, type flow application is actually many-to-many, since it supports functional arguments which are themselves function invocations. A function invocation returns a set of execution strands, each containing a potentially different argument type.

To understand type flow application, it helps to remember this representational invariant: the "result" of type flow application is always a braid. This means that the type flow analysis of any expression is a braid—be it a function application such as `(push item stack)`, a variable such as `object`, or even a primitive value such as `10`.

Generating the braid representing the structure of a function is, at one level, very simple. Let the set of parameters to the function **F** be $p_1 \ldots p_n$, and its body be the sequence of function applications $f_1 \ldots f_n$. Begin by generating an initial braid with the structure:

$$Br_0\,[St_{01}\{\ \mathrm{T}\ \langle\ p_1{:}\mathrm{v}_1\ \ldots\ p_n{:}\mathrm{v}_n\ \rangle\ \mathrm{v}_m\}]$$

This braid simply initializes the constraint environment with the parameters to the function and provides them with unique type variables. Starting with this braid, each of the functions $f_1 \ldots f_n$ in the body will be analyzed in turn by the type flow analyzer function, $\mathcal{TFA}$. This function takes a braid, and a function application, and returns a new braid that extends the execution paths represented by the argument braid with the potential execution paths represented by the function application:

$$\mathcal{TFA} : Br_i \times f_i \rightarrow Br_j$$

Given the function $\mathcal{TFA}$; determining the structure of **F** is simply a matter of evaluating:

$$\mathcal{TFA}(f_n, \mathcal{TFA}(f_{n-1}, \mathcal{TFA}(\ldots(\mathcal{TFA}(f_2, \mathcal{TFA}(f_1, Br_0))))))$$

In other words, invoke $\mathcal{TFA}$ on the initial braid and initial form in the function body to get the first braid, then invoke $\mathcal{TFA}$ on that first braid and the second form in the function body to get the second braid, and so forth.

3.4.1 The $\mathcal{TFA}$ mechanism

Imagine for a moment that $\mathcal{TFA}$ is midway through the processing of a function definition, and is invoked with a three-stranded braid $(Br_i\,[St_{i1}St_{i2}St_{i3}])$ and a function application (f_j). The job of $\mathcal{TFA}$ is to return a new braid that extends the three execution paths with all of the possible execution paths through the function application, and detect any type inconsistencies that might result from the attempt. This extension may result in the pruning, continuing, splitting, or merging of the three original execution strands.

For example, the execution strands St_{i1} and St_{i2} might each be extendable along five different strands in f_j, resulting in 10 execution paths, of which 3 are duplicates and pruned from the final braid. The third strand St_{i3} might be entirely infeasible along any of the execution paths in f_j, which prunes it from further consideration entirely. On the other hand, St_{i3} might have been feasible along one of the execution paths of f_j, but the extension might fail due to a type inconsistency. This example gives the flavor of the process: the

processing of each form may prune some paths, continue others, and split still others into multiple new paths.

The $\mathcal{TFA}$ function accomplishes this by analyzing the function application with respect to one of the execution strands and returning the braid representing the result of this analysis. This process of generating a braid from a single execution strand and a function application is termed *strand-level type flow application*, or $s\mathcal{TFA}$. The result of invoking $s\mathcal{TFA}$ on each of the execution strands is a set of new braids, which are composed together into a single braid which is returned as the result of the $\mathcal{TFA}$ application[5].

3.4.2 Strand-level type flow application

$s\mathcal{TFA}$ takes a function application and an execution strand and returns a braid representing the possible execution paths through the function application given the state of execution represented by the execution strand:

$$(f_i\ p_1\ p_2\ \ldots\ p_n) \times St_i \rightarrow Br_j$$

It begins by analyzing the arguments in the function application form to determine their type-level structure. Since the arguments to a function application may themselves be function applications, $\mathcal{TFA}$ is recursively invoked to perform this structural analysis. To accomplish this, $s\mathcal{TFA}$ performs the following transformation on the function application:

$$(f_1\ \mathcal{TFA}(p_1, Br_i[St_i])\ \mathcal{TFA}(p_2, Br_i[St_i])\ \ldots\ \mathcal{TFA}(p_n, Br_i[St_i]))$$

[5]This transition between $\mathcal{TFA}$ and $s\mathcal{TFA}$ is accomplished through the ESSIE language construct **with-strand**, which will be discussed in Section 4.4.2.

Note that a "singleton" braid is constructed around the current strand St_i since $\mathcal{TFA}$ requires a braid as an argument. Denote the result of applying $\mathcal{TFA}$ to parameter p_i as Br_i, and the above expression simplifies to:

$$(f_1\ Br_1\ Br_2\ \ldots\ Br_n)$$

Here each braid is subscripted with the corresponding parameter it was generated from. Each of these braids effectively "subsumes" the original strand St_i that $s\mathcal{TFA}$ started with, since they represent the additional execution paths after evaluation of the arguments to the function, but before the actual execution of the function. Conceptually, this representation embodies the idea that analysis of the arguments to the function could result in multiple execution paths leading to multiple result types for each argument. The task of $\mathcal{TFA}$ is now to sort out which combinations of argument execution strands are potentially consistent with each other, and generate a composite execution strand for each of those cases. This composite strand will then be used with the braid associated with f_1 to generate the result braid from application of $\mathcal{TFA}$ to f_1 and St_i[6].

To understand this process, consider the case of a function f_i with just two parameters. After applying $\mathcal{TFA}$ to its arguments, the result can be represented as follows:

$$(f_i\ Br_1\,[St_{11}\ St_{12}\ \ldots\ St_{1n}]\ Br_2\,[St_{21}\ St_{22}\ \ldots\ St_{2m}])$$

There are potentially $n \times m$ different potential control flows through the argument evaluation process into the function body f_i, corresponding to each

[6]This "sorting out" is accomplished through the ESSIE language construct with-arg-analyses, which is discussed in Section 4.4.2.

pairing of a strand St_{1j} from the first argument with a strand St_{2k} from the second argument. Each of these pairings is assessed for its *feasibility* (whether it could actually occur during execution) and its *consistency* (whether a feasible strand pairing would result in a structural inconsistency). Feasibility is assessed by composing the path conditions from the two strands together and determining if the resulting condition can be satisfied, while consistency is assessed by composing together the two constraint environments. The result types from the two execution strands indicate the actual type expressions to be associated with the arguments to the function application. Let's step through this in a bit more detail.

Taking a single pairing of execution strands St_{1j} and St_{2k} from the two argument braids Br_1 and Br_2, the situation can be represented as follows:

$$(f_i\ St_{1j}\{FC_j\ CE_j\ RT_j\}\ St_{2k}\{FC_k\ CE_k\ RT_k\})$$

This represents a potential function application of f_i to arguments of type RT_j and RT_k—*if* the feasibility condition formed from the composition of FC_j and FC_k is feasible, and *if* the composition of the constraint environments CE_j and CE_k is consistent. Let these compositions be denoted as $FC_{j\wedge k}$ and CE_{jok}[7]. Then if $FC_{j\wedge k}$ is feasible and CE_{jok} is consistent, the analysis of f_i becomes:

$$(f_i\ RT_j\ RT_k) \mid_{FC_{j\wedge k}\ CE_{jok}}$$

That is, $sTFA$ now retrieves the braid associated with the function f_i and applies it to the argument types RT_j and RT_k, in the context of the constraint

[7]The details of how feasibility conditions and constraint environments are composed are deferred until Section 3.4.5.

environment $CE_{j \circ k}$, and given the feasibility condition $FC_{j \wedge k}$. This process is called *braid application*.

3.4.3 Braid application

Braid application takes the braid representing a function (Br_f), a set of type expressions representing the arguments to the function ($tx_1 \ldots tx_n$), and an execution strand (St_i), and returns a new braid (Br_j) representing the extensions to the execution strand through the function body:

$$Br_f \times tx_1 \ldots tx_n \times St_i \rightarrow Br_j$$

In a manner analogous to $\mathcal{TFA}$, braid application applies each component execution strand in Br_f individually to the context represented by $tx_1 \ldots tx_n$ and St_i. To illustrate this process, consider an arbitrary strand in Br_f:

$$St_f\{FC_f \; CE_f \; RT_f\}$$

Applying this execution strand to the context results in a new composite execution strand that "continues" the execution path represented by St_i with St_f. Creating this composite involves the following: (1) assessment of the feasibility of the composite execution path; (2) assessment of the structural consistency of the composite execution path; and (3) generation of the new, feasible, structurally consistent composite execution path.

Assessing the feasibility of the composed execution path occurs in the following way. First, the type expressions $tx_1 \ldots tx_n$ associated with the ar-

guments are *substituted* for the type expressions associated with their corresponding parameters in the function strand's constraint environment, CE_f. This substitution results in a new constraint environment, $CE_{f'}$, as well as a new feasibility condition, $FC_{f'}$. A new feasibility condition results from this substitution since type expressions in constraint environments may share structure with type expressions in feasibility conditions.

The substituted feasibility condition $FC_{f'}$ is composed with the strand's feasibility condition FC_i to form a composite, $FC_{f'\wedge i}$. If $FC_{f'\wedge i}$ is unsatisfiable, then the processing of this strand terminates immediately. Otherwise, the type expressions $tx_1 \ldots tx_n$ associated with the arguments are *unified* with the type expressions associated with their corresponding parameters in the function's (original) constraint environment CE_f. If unification fails, then a structural inconsistency is signaled by $\mathcal{TFA}$.

If the argument type expressions successfully unify with the parameter type expressions, then all that remains is to create the composite execution strand. The composite feasibility condition is $FC_{f\wedge i}$, the constraint environment is the post-unification version of CE_f, and the new result type is RT_f. Since the result type expression may also potentially share structure with the type expressions in the constraint environment, the unification of the arguments with parameters may have affected its value.

The two-part substitution/unification strategy is required since unification alone might signal structural inconsistencies from infeasible composite paths.

Consider the following execution strand from the **size** function discussed in Chapter 1:[8]

$$St_{size}\{\ \text{(listp list[}v_1\text{]}\ \langle\ \text{object : list[}v_1\text{]}\ \rangle\ \text{integer}\}$$

During the processing of a function application like `(size "abc")`, the argument type of string corresponds to the parameter **object**, which has type list[v_1]. Simple unification of argument types to parameter types signals a structural inconsistency, even though the path is actually infeasible. *Substitution*, however, results in the following strand:

$$St_{size'}\{\ \text{(listp string)}\ \langle\ \text{object : string}\ \rangle\ \text{integer}\}$$

This strand $St_{size'}$ is infeasible, since the feasibility condition is unsatisfiable.

A final aspect of type flow application is brought out by consideration of the following function's definition and braid:

```
(defun add (x y)
  (cond ((integerp x)
         (+ x y))))
```

$$Br_{add}[St_1\{\ \text{(integerp integer)}\ \langle\ \text{x : integer; y : integer}\ \rangle\ \text{integer}\}]$$

The add function does nothing useful other than to construct a simple situation in which two arguments are constrained to the same type, yet only one argument is "protected" by a conditional test. Therefore, the expressions

[8]Note that in Chapter 1, the type predicate arguments in feasibility conditions were parameter names like `item` and `stack`. This was done to simplify the presentation, while the strand representation in this chapter accurately represents the results of type flow analysis.

`(+ 2 "string")` and `(+ "string" 2)` should result in different behaviors. In the case of `(+ 2 "string")`, the parameter x is passed an integer, and so the feasibility condition will be satisfied, and a type error will then be signalled when an attempt is made to unify the string argument of y to an integer. In the case of `(+ "string" 2)`, the strand will simply be infeasible, and an error will be eventually be signalled because no execution path through the function exists.[9] A straightforward reading of the above braid would appear to indicate that substitution of x *or* y as an integer would suffice to make the feasibility condition satisfiable.

Type flow analysis supports each of these behaviors by distinguishing between different instances of any kind type expressions, although the notation only uses subscripts to distinguish between different instances in the case of type variables. While this notational truncation causes no problems in most situations, certain functions like add do demonstrate its shortcomings. A more accurate representation of add would be the following:

$$Br_{add}[St_1\{\ (\text{integerp}\ \ \text{integer}_1)\ \langle\ \text{x} : \text{integer}_1;\ \text{y} : \text{integer}_2\ \rangle\ \text{integer}_3\}]$$

When individual instances of integers, as well as type variables, are subscripted, it is easy to see how type flow analysis processes functions with a structure similar to add correctly.

[9]Assuming the interpretation of cond clauses where "falling off" is prohibited. This is discussed further in Chapter 4.

3.4.4 Summary

So far, we have seen how type flow application involves (1) the analysis of the function's arguments, which results in a braid corresponding to each argument; (2) the pairwise combining of the component strands from each of these argument braids into a set of feasible composite argument/strand structures; (3) the processing of each of these argument/strand structures with each of the execution strands in the function's braid; and (4) the potential creation of a new, feasible, composite execution strand representing the type flow from the argument/strand structure through the function body along the execution path represented by the strand.

The above discussion deferred the description of composing feasibility conditions and constraint environments. The following section fills in these details. From there the description of ESSIE shifts from type flow application—or how to compose together pre-existing execution strands—to conditional processing, which describes how multiple execution strands are generated during analysis. Following conditional processing, the chapter ends with a description of recursion processing.

3.4.5 Feasibility condition and constraint environment composition

A prominent part of the manipulation of execution strands is the composition of their internal parts—specifically, the feasibility condition and the constraint environment. For two execution strands St_i and St_j, these compositions are denoted with "$\wedge$" for the feasibility condition and "o" for the

constraint environment—i.e. $FC_{i \wedge j}$ and CE_{ioj}. Different symbols are used since the two types of composition are performed in a different, though interdependent fashion: in general, composition of two feasibility conditions must be preceeded by composition of their associated constraint environments.

Constraint environment composition is a straightforward process of unifying corresponding type expressions from the two environments. What constitutes "corresponding type expressions" depends upon what part of type flow application is being performed. During braid application, for example, the two constraint environments consist of the same set of identifiers, though the identifiers may have different type expressions associated with them. In this case, corresponding type expressions are determined through their binding to the same identifier. This situation is illustrated in the following example composition of two constraint environments CE_i and CE_j into CE_{ioj}:

CE_i : x : integer; y : v_1; z : integer
CE_j : x : integer; y : string; z : v_3
CE_{ioj} : x : integer; y : string; z : integer

The resulting composite arises from the instantiation of v_1 to string, and v_3 to integer.

The second way to determine corresponding type expressions occurs when the argument type expressions are applied to the function's parameters. Here the corresponding type expressions are determined through the matching of argument type expression to parameter type expression, though these identifiers in general will be different.

Once the constraint environments have been composed, the feasibility conditions can be composed as well if necessary. Consider the following two execution strands built around the original constraint environments CE_i and CE_j from above:

St_i\{ (stringp v_1) ⟨ x : integer; y : v_1 ; z : integer ⟩ v_1\}
St_j\{ (not (stringp v_3)) ⟨ x : integer; y : string; z :v_3 ⟩ integer\}

Generating the composite feasibility condition begins by composing together the two constraint environments, and then concatenating the two feasibility conditions together. In this case, the composed constraint environment is $St_{i \circ j}$, as shown above. The composite feasibility condition $FC_{i \wedge j}$ is then:

$FC_{i \wedge j}$: (stringp string) (not (stringp integer))

The next step is to determine if the composite feasibility condition is satisfiable—if the predicate it represents can ever return `true`. The description of this assessment process is simplified by defining a few terms for the internal structure of a feasibility condition. First, a feasibility condition is made up of a sequence of *terms*, each of which consists of a (possibly negated) pairing of a type predicate function with a single type expression, called an *argument type expressions*. Furthermore, each type predicate function has a corresponding type expression—`stringp` is associated with string, `listp` is associated with list[v_i], etc. These are called *predicate type expressions*. Thus, $FC_{i \wedge j}$ consists of two terms, (*stringp* string) and (*not* (*stringp* integer))—and in the second term the predicate type expression is string, while the argument type expression is integer.

A feasibility condition is termed *satisfiable* if a set of substitutions exists over all the type variables appearing in the condition such that (1) all non-negated terms can successfully unify the predicate type expression with its argument's expression; and (2) all negated terms will unsuccessfully unify their predicate type expression against their argument's expression.

$FC_{i \wedge j}$ is a satisfiable feasibility condition, since its first term will unify and its second term will not. Unsatisfiable feasibility conditions generally occur when the results of unification in one term prevent successful unification in later terms, such as in the following examples:

FC_k : (stringp v_1) (not (stringp v_1))
FC_l : (stringp v_1) (listp v_1)

Satisfiability is assessed in a rather straightforward manner: unify all the non-negative terms first, and then check the remaining negated terms for contradictions (i.e. such as (*not* (*stringp* string))). It is interesting to note that this mechanism relies on an "open type world" assumption with respect to the type system. This means that a feasibility condition cannot be shown unsatisfiable by reasoning that every possible substitution has been negated. For example, the following feasibility condition is satisfiable under an open type world assumption, but unsatisfiable under a closed type world assumption (in a language with only string and integer types):

FC_m : (not (integerp v_1)) (not (stringp v_1))

The increased reasoning power available under a closed type world assumption comes at the cost of requiring complete knowledge about all the types in

the system. Since exploratory languages allow run-time definition of types, ESSIE does not assume complete type knowledge during its static analysis.

This concludes the discussion of composition of feasibility conditions and constraint environments in ESSIE. It also concludes the description of the facilities for type flow application—or sequential processing—of programs. The next section turns to the issue of conditional processing.

3.5 Conditional type flow processing

This section describes the type flow analysis of conditional statements. It expands upon the description of sequential processing presented in the last section by showing how feasibility conditions are initially generated from the test clauses in conditional forms. The following, final section of this chapter will complete the basic forms of type flow analysis by describing how iteration is accommodated in ESSIE.

Conditional type flow processing consists of four basic steps. The first step, *order dependency elimination*, performs a source level transformation of conditional test clauses into a form whereby their semantics no longer depends upon their order of occurrence within the conditional form. Order dependency elimination is necessary to allow the execution strands eventually generated from the conditional statement to be manipulated entirely independently.

The second step, *OR elimination*, transforms each conditional clause whose test contains an or logical connective into an equivalent *set* of conditional

clauses. The conditional tests in this new set of clauses correspond to each possible sequence of test expressions leading to satisfaction of the original conditional test clause. In effect, the `or` is transformed out of the test clause and into the structure of the cond clause as a whole. One purpose of OR elimination is to accomodate the short-circuiting nature of `or`. This feature allows different subsets of its arguments to be evaluated at run-time, where each subset leads to different type-level constraints on the body of the clause.

The third step, *feasibility condition generation*, takes the conditional tests from the last step and produces one or more feasibility conditions. This step must accommodate the fact that the arguments to type predicates are not always parameters, but may be arbitrary forms whose type-level structure is represented as a braid. Feasibility condition generation involves the processing of these pairings of type predicates and braids into actual feasibility conditions, which pair type predicates with type expressions.

The above three steps produce a feasibility condition and constraint environment for the execution of the conditional test body. The final step is straightforward: the body of the conditional is analyzed with respect to this feasibility condition and constraint environment through the sequential type flow analysis techniques. This is illustrated by the example processing of the function `double-reverse` in Section 3.3—the strands immediately following a conditional test reflect the result of order dependency elimination, OR elimination, and feasibility condition generation. The remainder of the conditional body is then processed with sequential methods. Each of the first three steps is now considered in detail.

3.5.1 Order dependency elimination

While execution strands are completely independent of each other, the same is not true for the conditional clauses that generate them. Consider the following common Lisp utility function, `first-if-list`[10]:

```
(defun first-if-list (obj)
  (cond ((listp obj)
         (first obj))
        (t
         obj)))
```

This provides one example of a general phenomenon: a `T` test clause itself provides no information on the type-level restrictions, though its context of use induces them. For `first-if-list`, the second clause has the type-level restriction that `obj` is not a list. The purpose of order dependency elimination is to make these implicit, contextual type-level restrictions explicit within the conditional test clause itself. This is accomplished by transforming the conditional form into one in which each conditional test clause is appended with the negation of all preceding test clauses. For `first-if-list`, this transformation would yield the following cond clause:

```
(cond ((listp obj)
       (first obj))
      ((and (not (listp obj)) t)
       obj))
```

Here the negation of the first test clause is concatenated with the second test clause. In general, the negation of each of the first n clauses is con-

[10] `First-if-list` is a more polymorphic version of `car` (or `first`), in that `first-if-list` is defined over objects of any type. It is useful in situations where an object may consist of one or more components, and it is inefficient and/or laborious to convert a single component to a list consisting of one element.

catenated with the $(n+1)$st clause. Note that logical expression simplification mechanisms are present in ESSIE to reduce expressions like `(and (not (listp obj)) t)` to `(not (listp obj))`, though the unreduced form is shown in this example for expository purposes. These order-independent conditional forms are then passed on to the next phase of conditional type flow processing, where `OR` forms are eliminated.

3.5.2 OR elimination

The presence of `or` forms in conditional test clauses raises interesting issues because different type-level constraints may be placed on identifiers based upon the particular sequence of forms evaluated before the form succeeds. Consider the function `empty-stack`[11] in Figure 9.

Processing functions like `empty-stack` poses two significant problems for conditional type flow analysis. First, such conditional tests use type predicates as *screens*—the `listp` function "protects" the `endp` function from being invoked on anything but list types, just as `vectorp` protects `aref`. One task of the analysis mechanism is to correctly model this screening behavior, so that type flow analysis does not raise an "ersatz" error—for example, that in `empty-stack`, a list object could be passed to the `aref` function.

A second problem in the analysis of `empty-stack` is that in the body of the first conditional clause, stack could be bound to either a list or a vector,

[11] `Empty-stack` would normally be defined as simply the logical expression in the first conditional test form. However, for ease of analysis, ESSIE performs a truth-preserving source-level transformation of such "stand-alone" logical expressions into a corresponding conditional format. The current definition of `empty-stack` would be the result of such a transformation.

• *Definition of* empty-stack*:*

```
(defun empty-stack (stack)
  (cond ((or (and (listp stack) (endp stack))
             (and (vectorp stack) (= 0 (aref 0 stack))))
         t)

        (t
          nil)))
```

• *The (unsimplified) order-independent conditional form in* empty-stack*:*

```
(cond ((or (and (listp stack) (endp stack))
           (and (vectorp stack) (= 0 (aref 0 stack))))
       t)

      ((and t
            (not (or (and (listp stack) (endp stack))
                     (and (vectorp stack) (= 0 (aref 0 stack))))))
       nil))
```

Figure 9: Conditional analysis in EMPTY-STACK

which seems to require the prohibited disjunctive "list∧vector" type. OR elimination both solves the problems of checking screens and eliminates the need for disjunctive types.

In OR elimination, conditional clauses are rewritten so that no or forms appear in the conditional tests. Instead, the disjunctions are implicit in the structure of the conditional clause itself. Conceptually, the rewritten form corresponds to making multiple conditional clauses with identical bodies for each term in the disjunction. In reality, the process is a bit more complicated, since OR elimination also checks to make sure that any type-level screening succeeds.

Testing for screen failure involves generating for each conditional test a set of *satisfaction sequences*—a set of forms that must evaluate to true in order for the conditional test as a whole to evaluate to true. For conjunctions (and forms) without internal disjunctions, the satisfaction sequence is simply the and form itself. For disjunctions without internal disjunctions, the satisfaction sequences consist of the first term, the first term negated plus the second term, the first and second terms negated plus the third term, and so forth:

$$(or\ t_1\ t_2 \ldots t_n) \rightarrow \{t_1\}\ \{(not\ t_1)\ t_2\} \ldots \{(not\ t_1)\ (not\ t_2) \ldots t_n\}$$

Since logical expressions can be arbitrarily nested, determination of the satisfaction sequence set is a recursive, bottom up process. In addition, logical expression simplification techniques are used to eliminate double negations, and distribute negations over internal terms.

Each member of the satisfaction sequence set becomes, in effect, a new

conditional test for a duplicate of the original conditional test's body. For example, from the first conditional test form in `empty-stack`:

```
(or (and (listp stack) (endp stack))
    (and (vectorp stack) (= 0 (aref 0 stack))))
```

the following set of satisfaction sequences are generated:

```
(listp stack) (endp stack)

(not (listp stack)) (vectorp stack) (aref 0 stack))

(listp stack) (not (endp stack)) (vectorp stack) (= 0 (aref 0 stack))
```

Figure 10 illustrates the generation of the set of satisfaction sequences for the second clause in `empty-stack`. Some logical expression simplification is also illustrated in this figure.

Figure 11 shows a representation of the conditional form following OR elimination[12]. The OR eliminated form of `empty-stack` also seems to illustrate a potential problem: the third and sixth conditional clauses contain invocations of both `null` and `aref` on `stack`. This would appear to simultaneously constrain `stack` to both a list (through `null`) and a vector (through `aref`)—a clear structural inconsistency that does not, however, appear in the original source code of Figure 9. This problem, fortunately, is illusory since the corresponding conditional test, which contains both `(listp stack)` and `(vectorp stack)`, is infeasible.

The conditional form has now been massaged into a form that reveals exactly which forms will be processed and in what order along all the potential

[12]ESSIE does not represent the result of OR elimination entirely as a source-level transformation of the conditional clause, but the differences are simply implementation details.

- *Original order-independent test clause:*

```
(and t
     (not (or (and (listp stack) (endp stack))
              (and (vectorp stack) (= 0 (aref 0 stack))))))
```

- *Simplified expression ready for satisfaction sequence generation:*

```
(and (or (not (listp stack) (not (endp stack))))
     (or (not (vectorp stack)) (not (= 0 (aref 0 stack)))))
```

- *Generated satisfaction sequences:*

```
(not (listp stack)) (not (vectorp stack))

(listp stack) (not (endp stack)) (not (vectorp stack))

(listp stack) (not (endp stack)) (vectorp stack)
(not (= 0 (aref 0 stack)))

(not (listp stack)) (vectorp stack) (not (= 0 (aref 0 stack)))
```

Figure 10: Satisfaction sequence generation for EMPTY-STACK

• *The transformed conditional for* empty-stack*:*

```
(cond ((and (listp stack) (endp stack))
       t)

      ((and (not (listp stack)) (vectorp stack)
            (= 0 (aref 0 stack)))
       t)

      ((and (listp stack) (vectorp stack)
            (not (endp stack)) (= 0 (aref 0 stack)))
       t)

      ((and (not (listp stack)) (not (vectorp stack)))
       nil)

      ((and (listp stack) (not (vectorp stack)) (not (endp stack)))
       nil)

      ((and (listp stack) (vectorp stack)
            (not (endp stack)) (not (= 0 (aref 0 stack))))
       nil)

      ((and (not (listp stack)) (vectorp stack)
            (not (= 0 (aref 0 stack))))
       nil))
```

Figure 11: OR elimination results for the conditional in EMPTY-STACK

execution paths represented by the conditional test clauses. The next section describes exactly how these processed conditional test clauses are transformed into the actual feasibility conditions of an execution strand.

3.5.3 Feasibility condition generation

Feasibility condition generation consists of transforming the conditional test clauses into the actual predicate-type expression pairs found in feasibility conditions. Given the types of examples seen thus far, there might appear to be a straightforward correspondence between type predicates in conditional tests and those in feasibility conditions. In fact, this correspondence is to a great extent an artifact of the examples—caused in particular by the fact that the arguments to the type predicates were parameters to the function itself. Unfortunately, this is not generally the case—the arguments to type predicates may be identifiers other than parameters, or the arguments may be themselves function applications.

To illustrate this, consider the function `stack-transition-point-p` in Figure 12. This function detects if pushing an additional item onto a stack will result in its being dynamically re-represented. In order to do so, the type predicate `vectorp` is passed not a parameter, but a function application, whose type-level analysis of necessity must be a braid, not a type expression as the feasibility condition representation demands. In fact, analyzing the argument to a type predicate always returns a braid, even if the argument is a parameter, and a major task of feasibility condition generation is the processing of these pairings of type predicates and braids into a new form, where type predicates are paired with type expressions.

• *Definition of* stack-transition-point-p*:*

```
(defun stack-transition-point-p (stack)
  (cond ((and (listp stack)
              (vectorp (push 'nil stack)))
         t)

        (t
         nil)))
```

• *Its* cond *form after order-dependency elimination:*

```
(cond ((and (listp stack)
            (vectorp (push 'nil stack)))
       t)

      ((and t (not (and (listp stack)
                        (vectorp (push 'nil stack)))))
       nil))
```

• *The* cond *form after OR-elimination:*

```
(cond ((and (listp stack) (vectorp (push 'nil stack)))
       t)

      ((and (listp stack) (not (vectorp (push 'nil stack))))
       nil)

      ((and (not (listp stack)) (vectorp (push 'nil stack)))
       nil)

      ((and (not (listp stack)) (not (vectorp (push 'nil stack))))
       nil))
```

Figure 12: Preliminary conditional processing for STACK-TRANSITION-POINT-P

Feasibility condition generation involves three major steps. First, the argument to each type predicate term in the conditional test clause is analyzed to get its corresponding braid. The remaining terms are analyzed in their entirety to get their corresponding braids. Note that each component execution strand in the type predicate argument braids provides, via its result type, a potential type expression for the arguments to the type predicates, as well as accompanying feasibility conditions and constraint environments.

Second, the feasible and consistent cross product of these "braided conditional terms" is formed. The effect of this process is to transform the representation of the conditional clause from one in which multiple execution paths are represented into an expanded set of conditional clauses sharing the same conditional body, but whose conditional test represents only a single execution path.

The final step of the process is to transform these cross-product forms into the actual strand representation. This three-step feasibility condition generation algorithm will be first described in general, and then followed by a more detailed description involving the sequence of manipulations occurring during the processing of a representative function definition.

Feasibility condition generation begins with a satisfaction sequence, which is a sequence of type predicates (tp_i) with arguments (f_i), and non-type predicates (also f_i):

$$(tp_1 \ \ f_1) \ \ f_2 \, (tp_2 \ \ f_3) \ldots (tp_n \ \ f_n)$$

The first step is to invoke $\mathcal{TFA}$ on each of $f_1 \ldots f_n$ to determine their corresponding braid. (For clarity's sake, the following steps will be described for a

satisfaction sequence consisting of simply two type predicate elements, whose argument braids have only two strands. The mechanism is of course general for n-stranded braids, and the extensions to handle non-type predicate conditional terms are minor implementation details.) Invoking $\mathcal{TFA}$ leads to the following situation, where tp_1 is paired with Br_1, and tp_2 is paired with Br_2:

$$(tp_1\ Br_1\,[St_{11}\{FC_{11}CE_{11}RT_{11}\}St_{12}\{FC_{12}CE_{12}RT_{12}\}])$$
$$(tp_2\ Br_2\,[St_{21}\{FC_{21}CE_{21}RT_{21}\}St_{22}\{FC_{22}CE_{22}RT_{22}\}])$$

The next step is to form the strand cross product, which represents all possible execution path combinations during evaluation of the arguments to the type predicates:

$$(tp_1\ \ St\{FC_{11\wedge 21}CE_{11\circ 21}RT_{11}\})(tp_2\ \ St\{FC_{11\wedge 21}CE_{11\circ 21}RT_{21}\})$$
$$(tp_1\ \ St\{FC_{11\wedge 22}CE_{11\circ 22}RT_{11}\})(tp_2\ \ St\{FC_{11\wedge 22}CE_{11\circ 22}RT_{22}\})$$
$$(tp_1\ \ St\{FC_{12\wedge 21}CE_{12\circ 21}RT_{12}\})(tp_2\ \ St\{FC_{12\wedge 21}CE_{12\circ 21}RT_{21}\})$$
$$(tp_1\ \ St\{FC_{12\wedge 22}CE_{12\circ 22}RT_{12}\})(tp_2\ \ St\{FC_{12\wedge 22}CE_{12\circ 22}RT_{22}\})$$

The implications of the above become more clear by factoring out the common feasibility condition and constraint environment from each strand:

$$(tp_1\ \ RT_{11})\ \ (tp_2\ \ RT_{21})|_{FC_{11\wedge 21}CE_{11\circ 21}}$$
$$(tp_1\ \ RT_{11})\ \ (tp_2\ \ RT_{22})|_{FC_{11\wedge 22}CE_{11\circ 22}}$$
$$(tp_1\ \ RT_{12})\ \ (tp_2\ \ RT_{21})|_{FC_{12\wedge 21}CE_{12\circ 21}}$$
$$(tp_1\ \ RT_{12})\ \ (tp_2\ \ RT_{22})|_{FC_{12\wedge 22}CE_{12\circ 22}}$$

In other words, the initial pairings of type predicates with braids have been reduced to pairings of type predicates with type expressions—precisely the form needed for feasibility conditions. In fact, the final feasibility condition is created by simply appending the sequence of type predicates and result type pairs on to the feasibility condition in the corresponding strand:

$$FC_{11\wedge 21\wedge(tp_1\ \ RT_{11})\wedge(tp_2\ \ RT_{21})}$$
$$FC_{11\wedge 22\wedge(tp_1\ \ RT_{11})\wedge(tp_2\ \ RT_{22})}$$
$$FC_{12\wedge 21\wedge(tp_1\ \ RT_{12})\wedge(tp_2\ \ RT_{21})}$$
$$FC_{12\wedge 22\wedge(tp_1\ \ RT_{12})\wedge(tp_2\ \ RT_{22})}$$

This process shows that from an initial satisfaction sequence with two type predicate forms, where both arguments evaluate to a braid with two strands, there may be as many as four distinct execution paths into the body of the conditional clause. Each of these strands would be independently followed through the processing of the clause body.

This general description of the feasibility condition generation algorithm left out a few details, such as the detection of infeasible strands and the elimination of duplicates. The next section illustrates this algorithm in a bit more detail as a representative function is processed.

An example of complex conditional processing

Figures 12, 13, and 14 together illustrate how conditional tests are processed from beginning to end. Figure 12 shows the results of the first two stages—order dependency elimination and OR elimination. As illustrated by the figure, these stages can be viewed as syntactic manipulation of the source code into a form amenable to feasibility condition generation.

Figure 13 illustrates the initial phases of feasibility condition generation for one of the satisfaction sequences generated from `stack-transition-point-p`.

- *One satisfaction sequence from* `stack-transition-point-p`*:*

```
(and (listp stack) (vectorp (push 'nil stack)))
```

- *Braid generation through analysis of the predicate arguments:*

$$(\textit{listp}\ Br_1\,[St_{11}\{\ \mathrm{T}\ \langle\ \mathrm{stack} : \mathrm{v}_1\ \rangle\ \mathrm{v}_1\}])$$
$$(\textit{vectorp}\ Br_2 \begin{bmatrix} St_{21}\{\ (\mathrm{listp\ list}[\mathrm{v}_2])\ \langle\ \mathrm{stack} : \mathrm{list}[\mathrm{v}_2]\ \rangle\ \mathrm{list}[\mathrm{symbol}]\} \\ St_{22}\{\ (\mathrm{listp\ list}[\mathrm{v}_3])\ \langle\ \mathrm{stack} : \mathrm{list}[\mathrm{v}_3]\ \rangle\ \mathrm{vector}[\mathrm{symbol}]\} \\ St_{23}\{\ (\mathrm{vectorp\ vector}[\mathrm{v}_4])\ \langle\ \mathrm{stack} : \mathrm{vector}[\mathrm{v}_4]\ \rangle\ \mathrm{vector}[\mathrm{symbol}]\} \end{bmatrix})$$

- *The cross product set of the individual strands:*

$(\textit{listp}\ St_{11}\{\ \mathrm{T}\ \langle\ \mathrm{stack} : \mathrm{v}_1\ \rangle\ \mathrm{v}_1\})$
$(\textit{vectorp}\ St_{21}\{\ (\mathrm{listp\ list}[\mathrm{v}_2])\ \langle\ \mathrm{stack} : \mathrm{list}[\mathrm{v}_2]\ \rangle\ \mathrm{list}[\mathrm{symbol}]\})$

$(\textit{listp}\ St_{11}\{\ \mathrm{T}\ \langle\ \mathrm{stack} : \mathrm{v}_1\ \rangle\ \mathrm{v}_1\})$
$(\textit{vectorp}\ St_{22}\{\ (\mathrm{listp\ list}[\mathrm{v}_3])\ \langle\ \mathrm{stack} : \mathrm{list}[\mathrm{v}_3]\ \rangle\ \mathrm{vector}[\mathrm{symbol}]\})$

$(\textit{listp}\ St_{11}\{\ \mathrm{T}\ \langle\ \mathrm{stack} : \mathrm{v}_1\ \rangle\ \mathrm{v}_1\})$
$(\textit{vectorp}\ St_{23}\{\ (\mathrm{vectorp\ vector}[\mathrm{v}_4])\ \langle\ \mathrm{stack} : \mathrm{vector}[\mathrm{v}_4]\ \rangle\ \mathrm{vector}[\mathrm{symbol}]\})$

Figure 13: Initial phases of feasibility condition generation.

- *Result type factoring:*

(*listp* v_1)	St_{11}\{ T ⟨ stack : v_1 ⟩ \}
(*vectorp* list[symbol])	St_{21}\{ (listp list[v_2]) ⟨ stack : list[v_2] ⟩ \}
(*listp* v_1)	St_{11}\{ T ⟨ stack : v_1 ⟩ \}
(*vectorp* vector[symbol])	St_{22}\{ (listp list[v_3]) ⟨ stack : list[v_3] ⟩ \}
(*listp* v_1)	St_{11}\{ T ⟨ stack : v_1 ⟩ \}
(*vectorp* vector[symbol])	St_{23}\{ (vectorp vector[v_4]) ⟨ stack : vector[v_4] ⟩ \}

- *Feasibility condition and constraint environment composition:*

(*listp* list[v_3])	
(*vectorp* vector[symbol])	St_{22}\{ (listp list[v_3]) ⟨ stack : list[v_3] ⟩ \}
(*listp* vector[v_4])	
(*vectorp* vector[symbol])	St_{23}\{ (vectorp vector[v_4]) ⟨ stack : vector[v_4] ⟩ \}

- *Final feasibility condition and constraint environment:*

St_{22}\{ (vectorp vector[symbol]) (listp list[v_3]) ⟨ stack : list[v_3] ⟩ \}

Figure 14: Final phases of feasibility condition generation.

In *braid generation*, *TFA* is invoked on each argument to the type predicates to determine its corresponding braid. Figure 13 shows the two braids, denoted Br_1 and Br_2 and their associated strands that result from this process.

In *strand cross product generation*, the type predicate/braid pairings are split into the set of all possible type predicate/execution strand pairings. The figure illustrates how copies of each strand from Br_1 are paired with copies of each strand from Br_2.

Figure 14 shows *result type factoring*, where the result types from the strands are propagated into the type predicate form, and the rest of the execution strand is extracted out. This leaves a set of type predicate/type expression pairs along with their corresponding execution strands. At this point, the feasibility of the new predicate/expression pairs are assessed. In the case of the example, the first strand cross product pair from St_{11} and St_{21} is discarded, since (`vectorp` list[symbol]) is infeasible.

Next, the execution strand's feasibility conditions and constraint environments are composed together, and the resulting feasibility is assessed. Here the strand cross product from St_{23} is found to be unsatisfiable from processing of the feasibility condition (`listp` vector[v_4]).

The last step consists of merely adding the type predicate/type expression pairings to the composite execution strand's feasibility condition, which, along with the constraint environment, becomes the set of contexts in which the body of the condition clause corresponding to the original satisfaction sequence

will be processed. In the current example, only a single execution strand composition from St_{22} remains.

Feasibility conditions are simplified in three principal ways during their generation and composition. First, T type predicate terms are always removed when any other type predicate terms are present. Second, redundant terms are removed. Third, constant terms (i.e. those argument type expressions that share no structure with the type expressions in the constraint environment, and thus can never be substituted for, or further constrained, during processing) are removed. In the example, the final feasibility condition term (`vectorp` vector[symbol]) is a constant term, which ESSIE simplifies away.

This concludes the description of conditional processing in ESSIE. The next section turns to the final form of control flow processing, iteration, which is currently implemented through recursion.

3.6 Iterative type flow processing

ESSIE supports general forms of iteration in terms of recursive functions[13], which presents two analysis-related problems. The first problem is that recursive calls violate the bottom-up assumption in type flow analysis—that analysis can proceed from the leaves of the call graph upward to the root. Put another way, recursive functions do not allow the straightforward determina-

[13]Specialized forms of iteration such as the mapping functions are also implemented in ESSIE.

tion of the structure of a function from the structure of its components, since the function is a component of itself.

By introducing a loop into the call graph, recursive functions prevent the straightforward approach to processing implied by the description of the $\mathcal{TFA}$ algorithm. In general, type flow analysis handles recursion in the following way. When a recursive function call is encountered during the normal $\mathcal{TFA}$-based analysis of a function, the analysis mechanism obviously cannot look up the braid structure of the function, since that is what it is in the process of creating. However, $\mathcal{TFA}$ expects a braid to result from the analysis of a function call. To resolve this dilemma, the recursion handling mechanism returns a *contextual braid* for the recursive invocation containing execution strands that contain special information. First, the constraint environment of these strands includes not only the normal environment bindings uncovered during execution, but an additional binding providing type-level information about the invocation of the recursive function. Second, the result type of these strands consists of a special kind of type expression, called a *contextual type variable*. This variable serves as a kind of "flag," indicating where the recursive invocation occurred during processing. This pointer to the recursive context is later "resolved," once more information about the structure of the function has been determined.

The creation of contextual braids for a recursive function invocation satisfies the short term goal of allowing $\mathcal{TFA}$-based analysis to proceed, but appears to subvert two important goals the mechanism. First, normal type flow application checks to ensure that the arguments to a function are consistent

with the type-level constraints of the parameters. In this case, no consistency checking is performed since there is not yet complete parameter information for the recursive function. Second, rather than a set of strands including the actual return types from application of the function, contextual type variables are returned, which provide no real type level constraints.

To overcome these difficulties, the processing of recursive functions requires some additional analysis of this *intermediate* braid returned from the first pass over the function definition. This *recursive post-processing* exists to make sure that argument and parameter constraints are consistent, and that contextual type variables are "replaced," when necessary, by the actual types returned by the function. The following description of this processing is broken down into two cases: tail-recursive functions and non-tail recursive functions.

3.6.1 Tail recursion and context variable elimination

Figure 15 illustrates a tail-recursive function, which takes an `item` and a `stack` (implemented as either a list or a vector), and returns the symbol `t` if `item` is contained within the `stack`, and the symbol `nil` otherwise.

Figure 16 illustrates the set of four execution strands that result from the contextual braid processing techniques described above. The first two strands represent the non-recursive and recursive cases, respectively, when `stack` is a list, while the third and fourth strands represent the cases when `stack` is a vector. Although there are six branches in the function definition, only four execution strands result, since the first two branches of the first interior

```
(defun in-stk-p (item stack)
  (cond ((listp stack)
         (cond ((endp stack)
                nil)
               ((equal (first stack) item)
                t)
               (t
                (in-stk-p item (cdr stack)))))

        ((vectorp stack)
         (cond ((= 0 (aref 0 stack))
                nil)
               ((equal item (aref (aref 0 stack) stack))
                t)
               (t
                (in-stk-p item (make-popped-stack stack)))))))
```

Figure 15: The definition of IN-STK-P

conditional are structurally identical at the type level, as are the first two branches of the second interior conditional. Context variables are denoted by the prefix *cv*.

To understand the issues of recursion better, it may help to think a little about the nature of (tail) recursive functions. They consist of one or more *base* cases, implemented as non-recursive branches of a conditional clause, and one or more *inductive* cases, implemented as recursive branches. A terminating recursive function begins execution by following the recursive branches 0 or more times, but must ultimately take a non-recursive branch to prevent an infinite loop. Thus, the type of object returned from the recursive call inside the function will be one of the types returned from execution of the base cases,

• *Braids from processing* `in-stk-p` *before recursive post-processing*

$$
Br_1 \left[\begin{array}{ll}
St_{11} & \left\{ \begin{array}{ll} FC: & (\textit{listp}\ \mathsf{list}[\mathsf{v}_2]) \\ CE: & \textit{item}: \mathsf{v}_1; \textit{stack}: \mathsf{list}[\mathsf{v}_2] \\ RT: & \mathsf{symbol} \end{array} \right. \\
St_{12} & \left\{ \begin{array}{ll} FC: & (\textit{listp}\ \mathsf{list}[\mathsf{v}_4]) \\ CE: & \textit{item}: \mathsf{v}_3; \textit{stack}: \mathsf{list}[\mathsf{v}_4]; \\ & \textit{in-stk-p}: (\textit{item}: \mathsf{v}_3; \textit{stack}: \mathsf{list}[\mathsf{v}_4]; \textit{result}: \mathsf{cv}_1) \\ RT: & \mathsf{cv}_1 \end{array} \right. \\
St_{13} & \left\{ \begin{array}{ll} FC: & (\textit{vectorp}\ \mathsf{vector}[\mathsf{v}_6]) \\ CE: & \textit{item}: \mathsf{v}_5; \textit{stack}: \mathsf{vector}[\mathsf{v}_6] \\ RT: & \mathsf{symbol} \end{array} \right. \\
St_{14} & \left\{ \begin{array}{ll} FC: & (\textit{vectorp}\ \mathsf{vector}[\mathsf{v}_8]) \\ CE: & \textit{item}: \mathsf{v}_7; \textit{stack}: \mathsf{vector}[\mathsf{v}_8]; \\ & \textit{in-stk-p}: (\textit{item}: \mathsf{v}_7; \textit{stack}: \mathsf{vector}[\mathsf{v}_8]; \textit{result}: \mathsf{cv}_2) \\ RT: & \mathsf{cv}_2 \end{array} \right.
\end{array} \right]
$$

Figure 16: Intermediate braid for IN-STK-P

or non-recursive conditional branches. In the case of `in-stk-p`, that should be a symbol—`t` or `nil`, regardless of whether stack is a list or a vector. From this reasoning, one should expect that the analysis of `in-stk-p` should generate a braid with two strands, one for the list stack, and one for the vector stack.

However, the analysis results just prior to recursive post-processing illustrated in Figure 16 contain four strands, corresponding to the six execution paths through the function when contextual braids are returned.

Type flow analysis performs *context type variable removal* as a part of the process of converting this set of contextual execution strands into the braid that actually represents its structure. Context type variable removal consists of transforming any execution strand containing a context type variable into one or more execution strands without context type variables. Conceptually,

what happens is a sort of "loop unrolling"—-where the execution strand is "extended" to include the execution of the non-recursive execution strands in the function definition.

Figure 17 illustrates this process for the first execution strand containing a context variable in Figure 15 The process begins by collecting together all the non-tail recursive execution strands (those without context variables as their result types). In `in-stk-p`, there are two non-tail recursive execution strands. The context type variable removal mechanism will attempt to change the recursive execution strand to one or more non-recursive strands by processing it with each of the non-contextual strands in turn. Conceptually, this processing corresponds to creating a new execution path that corresponds to the extension of the contextual execution path with the non-contextual execution path. This *contextual execution path extension* can be broken down into the following four steps:

1. *Parameter-argument and result matching.* The first task is to match up the type expressions from the two execution strands. This involves matching the parameters to the function in the non-recursive case to the arguments in the function invocation expression in the constraint environment of the recursive strand. In addition, the result type of the function (which corresponds to a context type variable) is matched to the result type of the non-recursive execution strand. This matchup presents the following subquestions: (1) Does this match-up represent a *valid*

• *Example contextual execution strand from* `in-stk-p`*:*

$$St_{12}\left\{\begin{array}{ll} FC: & (\textit{listp}\ \text{list}[v_4]) \\ CE: & \textit{item}: v_3; \textit{stack}: \text{list}[v_4]; \\ & \textit{in-stk-p}: (\textit{item}: v_3; \textit{stack}: \text{list}[v_4]; \textit{result}: cv_1) \\ RT: & cv_1 \end{array}\right\}$$

• *Potential non-recursive extensions:*

$$St_{11}\left\{\begin{array}{ll} FC: & (\textit{listp}\ \text{list}[v_2]) \\ CE: & \textit{item}: v_1; \textit{stack}: \text{list}[v_2] \\ RT: & \text{symbol} \end{array}\right\}$$

$$St_{13}\left\{\begin{array}{ll} FC: & (\textit{vectorp}\ \text{vector}[v_6]) \\ CE: & \textit{item}: v_5; \textit{stack}: \text{vector}[v_6] \\ RT: & \text{symbol} \end{array}\right\}$$

• *Parameter/argument matching of function invocation structure with* first *non-recursive extension:*

item : v_3 ; stack : list[v_4] ; result : cv_1

St_{11}\{ (listp list[v_2]) ⟨ item : v_1 ; stack : list[v_2] ⟩ symbol\}

• *Substitution and feasibility check of example extension:*

$St_{11'}$\{ (listp list[v_4]) ⟨ item : v_3 ; stack : list[v_4] ⟩ symbol\}

• *Unification and context variable instantiation of function invocation structure:*

item : v_1 ; stack : list[v_4] ; result : symbol

• *Resolved version of contextual execution strand after extension:*

$$St_{12}\left\{\begin{array}{ll} FC: & (\textit{listp}\ \text{list}[v_4]) \\ CE: & \textit{item}: v_3; \textit{stack}: \text{list}[v_4] \\ RT: & \text{symbol} \end{array}\right\}$$

Figure 17: Context variable removal for an example execution strand

composite execution path? (2) If valid, does it represent a *consistent* execution path? (3) What is the non-contextual result type?

2. *Assess the validity of the extended execution strand.* The validity of the match-up can be assessed by referring to the path predicates: does the replacing of the "parameters" to the function (in the constraint environment of the recursive execution strand) by arguments (the corresponding type expressions in the constraint environment of the non-recursive execution strand) still represent a valid (i.e. conceivable) execution path? This question is answered by the same substitution/feasibility assessment procedure employed in *sTFA*. If the substituted feasibility condition is unsatisfiable, then this particular extension cannot occur in reality. Such a situation is illustrated in Figure 18, where the recursive strand corresponding to a list stack is matched against the non-recursive strand corresponding to a vector stack.

3. *Assess the consistency of the extended execution strand.* If the match-up is valid, indicating that the proposed extension of the recursive strand with the non-recursive strand could occur, the next step is to see whether any structural inconsistencies would result. In ESSIE, this is accomplished through unifying the matched parameters, arguments, and return type expressions. If this unification process fails, then a structural inconsistency exists.

4. *Replace the recursive strand by the set of non-recursive extended strands.* If the extension is both valid and consistent, it results in a new execu-

- *Example contextual execution strand from* `in-stk-p`*:*

$$St_{12}\left\{\begin{array}{ll} FC: & (\textit{listp}\ \text{list}[v_4]) \\ CE: & \textit{item}: v_3;\ \textit{stack}: \text{list}[v_4]; \\ & \textit{in-stk-p}: (\textit{item}: v_3;\ \textit{stack}: \text{list}[v_4];\ \textit{result}: cv_1) \\ RT: & cv_1 \end{array}\right\}$$

- *Potential non-recursive extensions:*

$$St_{11}\left\{\begin{array}{ll} FC: & (\textit{listp}\ \text{list}[v_2]) \\ CE: & \textit{item}: v_1;\ \textit{stack}: \text{list}[v_2] \\ RT: & \text{symbol} \end{array}\right\}$$

$$St_{13}\left\{\begin{array}{ll} FC: & (\textit{vectorp}\ \text{vector}[v_6]) \\ CE: & \textit{item}: v_5;\ \textit{stack}: \text{vector}[v_6] \\ RT: & \text{symbol} \end{array}\right\}$$

- *Parameter/argument matching of function invocation structure with* second *non-recursive extension:*

item : v_3 ; stack : list[v_4] ; result : cv_1

St_{13}\{ (vectorp vector[v_6]) ⟨ item : v_5 ; stack : vector[v_6] ⟩ symbol\}

- *Substitution and feasibility check of example extension:*

$St_{13'}$\{ (vectorp list[v_4]) ⟨ item : v_3 ; stack : list[v_4] ⟩ symbol\}

- *This substituted strand is infeasible, so proposed extension cannot occur.*

Figure 18: Attempting an infeasible contextual extension

tion strand that contains no context variables. Applying the above steps to each of the potential non-recursive extensions to the recursive execution strand may result in several new extended, non-recursive execution strands.

In the case of `in-stk-p`, each of the contextual execution strands has exactly one feasible and consistent noncontextual extension, which leads to four noncontextual strands. After removing duplicate strands, the resulting

double stranded braid is precisely the structure "expected" by the intuitive analysis, as shown below:

$$Br_1 \left[\begin{array}{ll} St_{11} \left\{ \begin{array}{ll} FC: & (\textit{listp}\ \mathsf{list}[v_2]) \\ CE: & \textit{item}: v_1; \textit{stack}: \mathsf{list}[v_2] \\ RT: & \mathsf{symbol} \end{array} \right\} \\ St_{13} \left\{ \begin{array}{ll} FC: & (\textit{vectorp}\ \mathsf{vector}[v_6]) \\ CE: & \textit{item}: v_5; \textit{stack}: \mathsf{vector}[v_6] \\ RT: & \mathsf{symbol} \end{array} \right\} \end{array} \right]$$

3.6.2 Non-tail recursion

Tail-recursive functions present both the problem of removing the context variables from the final structure of the function, as well as the problem of ensuring that the internal recursive invocation of the function is consistent with its structure. Non-tail recursive functions pose only the second problem, since the context variables tend not to show up in the execution strands used to form the final braid. This is illustrated in Figures 19 and 20, which show the definition and intermediate braid for a function computing the size of a stack.

`Stack-size` is not tail-recursive, since the recursive call is not the last function invocation of any conditional branch (the + function is the last function invocation). The "removal" of the context variable occurs when the context variable returned from the recursive call is constrained to an integer. This constraint follows from the role the context variable plays as an argument to the + function, which requires an integer argument.

```
(defun stack-size (stack)
  (cond ((listp stack)
         (cond ((endp stack)
                0)
               (t
                (+ 1 (stack-size (cdr stack))))))

        ((vectorp stack)
         (cond ((= 0 (aref 0 stack))
                0)
               (t
                (+ 1 (stack-size (make-popped-stack stack)))))))))
```

Figure 19: The definition of STACK-SIZE

• *Braids from processing* `stack-size` *before recursive post-processing*

$$
Br_1 \left[\begin{array}{l}
St_{11} \left\{ \begin{array}{ll} FC: & (\textit{listp}\ \text{list}[v_1]) \\ CE: & \textit{stack} : \text{list}[v_1] \\ RT: & \text{integer} \end{array} \right\} \\
St_{12} \left\{ \begin{array}{ll} FC: & (\textit{listp}\ \text{list}[v_2]) \\ CE: & \textit{stack} : \text{list}[v_2]; \\ & \textit{stack-size} : (\textit{stack} : \text{list}[v_2]; \textit{result} : \text{integer}) \\ RT: & \text{integer} \end{array} \right\} \\
St_{13} \left\{ \begin{array}{ll} FC: & (\textit{vectorp}\ \text{vector}[v_3]) \\ CE: & \textit{stack} : \text{vector}[v_3] \\ RT: & \text{integer} \end{array} \right\} \\
St_{14} \left\{ \begin{array}{ll} FC: & (\textit{vectorp}\ \text{vector}[v_4]) \\ CE: & \textit{stack} : \text{vector}[v_4]; \\ & \textit{stack-size} : (\textit{stack} : \text{vector}[v_4]; \textit{result} : \text{integer}) \\ RT: & \text{integer} \end{array} \right\}
\end{array} \right]
$$

Figure 20: Intermediate braid for STACK-SIZE

Figure 19 shows the execution strands collected at the end of processing the function body. Although context variables do not appear in the constraint environments or result types of these strands, there remains the question of assessing the consistency of the internal use of the function with its external structure. This assessment is performed with the same mechanisms used for contextual execution path extension for tail-recursive functions, except this time there is no need to actually instantiate the contextual variable. The goal of this process is to ensure that for each noncontextual execution strand extension possibility, the argument and result types are compatible. An example of this process is illustrated in 21.

Figure 22, on the other hand, illustrates a function with a structural error, and how the extension process finds it. This `bad-stack-size` function returns `nil` as one of its base cases, rather than `0`.

Figure 23 shows the braid resulting from the analysis of the function but before recursive post-processing. Although the entire function definition has been processed at this point without finding any structural inconsistencies, one still exists, since one of the strands indicates that `stack-size-error` can return a symbol, which it cannot. This problem is uncovered when the internal function invocation is extended against the other noncontextual strands, as illustrated in Figure 24.

• *Example contextual execution strand from* `stack-size`*:*

$$St_{12}\left\{\begin{array}{ll} FC: & (\textit{listp}\ \text{list}[v_2]) \\ CE: & \textit{stack}: \text{list}[v_2]; \\ & \textit{stack-size}: (\textit{stack}: \text{list}[v_2]; \textit{result}: \text{integer}) \\ RT: & \text{integer} \end{array}\right\}$$

• *Potential non-recursive extensions:*

$$St_{11}\left\{\begin{array}{ll} FC: & (\textit{listp}\ \text{list}[v_1]) \\ CE: & \textit{stack}: \text{list}[v_1] \\ RT: & \text{integer} \end{array}\right\}$$

$$St_{13}\left\{\begin{array}{ll} FC: & (\textit{vectorp}\ \text{vector}[v_3]) \\ CE: & \textit{stack}: \text{vector}[v_3] \\ RT: & \text{integer} \end{array}\right\}$$

• *Parameter/argument matching of function invocation structure with* first *non-recursive extension:*

stack : list[v_2] ; result : integer

$$St_{11}\left\{\begin{array}{ll} FC: & (\textit{listp}\ \text{list}[v_1]) \\ CE: & \textit{stack}: \text{list}[v_1] \\ RT: & \text{integer} \end{array}\right\}$$

• *Substitution and feasibility check of example extension:*

$St_{11'}$\{ (listp list[v_2]) ⟨ stack : list[v_2] ⟩ integer\}

• *Unification and context variable instantiation of function invocation structure:*

stack : list[v_1] ; result : integer

• *Resolved version of contextual execution strand after extension:*

$$St_{12}\left\{\begin{array}{ll} FC: & (\textit{listp}\ \text{list}[v_1]) \\ CE: & \textit{stack}: \text{list}[v_1] \\ RT: & \text{integer} \end{array}\right\}$$

Figure 21: Context variable removal for an example execution strand

```
(defun bad-stack-size (stack)
  (cond ((listp stack)
         (cond ((endp stack)
                nil)
               (t
                (bad-stack-size (cdr stack)))))

        ((vectorp stack)
         (cond ((= 0 (aref 0 stack))
                0)
               (t
                (bad-stack-size (make-popped-stack stack)))))))
```

- *Note the first conditional arm returns* `nil`, *not* `0`.

Figure 22: The definition of BAD-STACK-SIZE

- *Braids from processing* `bad-stack-size` *before recursive post-processing*

$$
Br_1 \left[\begin{array}{ll}
St_{11} & \left\{ \begin{array}{ll} FC: & (\textit{listp}\ \text{list}[v_1]) \\ CE: & \textit{stack} : \text{list}[v_1] \\ RT: & \text{symbol} \end{array} \right\} \\
St_{12} & \left\{ \begin{array}{ll} FC: & (\textit{listp}\ \text{list}[v_2]) \\ CE: & \textit{stack} : \text{list}[v_2]; \\ & \textit{bad-stack-size} : (\textit{stack} : \text{list}[v_2]; \textit{result} : \text{integer}) \\ RT: & \text{integer} \end{array} \right\} \\
St_{13} & \left\{ \begin{array}{ll} FC: & (\textit{vectorp}\ \text{vector}[v_3]) \\ CE: & \textit{stack} : \text{vector}[v_3] \\ RT: & \text{integer} \end{array} \right\} \\
St_{14} & \left\{ \begin{array}{ll} FC: & (\textit{vectorp}\ \text{vector}[v_4]) \\ CE: & \textit{stack} : \text{vector}[v_4]; \\ & \textit{bad-stack-size} : (\textit{stack} : \text{vector}[v_4]; \textit{result} : \text{integer}) \\ RT: & \text{integer} \end{array} \right\}
\end{array} \right]
$$

Figure 23: Intermediate braid for BAD-STACK-SIZE

• *Example contextual execution strand from* `bad-stack-size`*:*

$$St_{12}\left\{\begin{array}{ll} FC: & (\textit{listp}\ \text{list}[v_2]) \\ CE: & \textit{stack}: \text{list}[v_2]; \\ & \textit{bad-stack-size}: (\textit{stack}: \text{list}[v_2]; \textit{result}: \text{integer}) \\ RT: & \text{integer} \end{array}\right\}$$

• *Potential non-recursive extensions:*

$$St_{11}\left\{\begin{array}{ll} FC: & (\textit{listp}\ \text{list}[v_1]) \\ CE: & \textit{stack}: \text{list}[v_1] \\ RT: & \text{symbol} \end{array}\right\}$$

$$St_{13}\left\{\begin{array}{ll} FC: & (\textit{vectorp}\ \text{vector}[v_3]) \\ CE: & \textit{stack}: \text{vector}[v_3] \\ RT: & \text{integer} \end{array}\right\}$$

• *Parameter/argument matching of function invocation structure with* first *non-recursive extension:*

stack : list[v_2] ; result : integer

$$St_{11}\left\{\begin{array}{ll} FC: & (\textit{listp}\ \text{list}[v_1]) \\ CE: & \textit{stack}: \text{list}[v_1] \\ RT: & \text{symbol} \end{array}\right\}$$

• *Substitution and feasibility check of example extension:*

St_{11}'{ (listp list[v_2]) ⟨ stack : list[v_2] ⟩ integer}

• *Unification and context variable instantiation of function invocation structure:*

ESSIE Error: integer **cannot unify with** symbol

Figure 24: Detecting a structural error during recursive post processing

3.6.3 Mutual recursion

The examples shown above were all examples of *single recursion*, where a function can invoke itself as well as other functions, but none of the other functions recursively invoked the original function. In *mutual recursion*, also known as indirect recursion, a function recursively calls itself through the invocation of a second function, rather than directly within its own body. Handling mutual recursion involves straightforward extensions to the mechanisms detailed above for single recursion.

The first extension required is the determination of the *strongly connected components* in the system. In other words, the system must first determine the sets of mutually recursive functions that must be analyzed together, since their definitions are mutually dependent. Determining the set of strongly connected components involves a straightforward traversal of the calling hierarchy of the system[14]

Once the sets of mutually recursive functions have been isolated, each of the functions in each of the sets is processed in much the same way as for single recursion, except that contextual braids are returned whenever an invocation of *any* of the functions in the strongly connected component set occurs. Thus, the braids for each function will contain function invocation structures for each of the functions in the mutually recursive set, rather than just for the function itself.

[14]This traversal is not yet implemented in the current version of ESSIE. Instead, the developer must provide the set of mutually recursive functions to ESSIE through the function **analyze-mutually-recursive-fns**, as described in Section 4.2.2.

The next step is the resolving of the contextual variables and assessment of structural consistency. Again, this occurs in much the same way as for single recursion, except that the resolution process involves the stepwise removal of each context variable across the entire set of functions, rather than within a single function.

3.7 Conclusion

This chapter has presented the fundamental mechanisms of ESSIE, which uses a type expression language in combination with representations for sequential, conditional, and iterative processing to implement the basic aspects of control and data flow.

This description concentrated on the particular mechanisms for manipulation of these representations using simple forms of control and data flow objects. The next chapter builds upon this discussion of mechanisms with a description of the high-level language constructs provided by ESSIE for the construction of type analysis systems. This next chapter shows how complex control and data flow can be built from the language constructs provided for the manipulation of these simple mechanisms.

Chapter 4

The Language of Type Flow Analysis

The last chapter presented a relatively abstract look at control and data flow in type flow analysis. It showed how to implement mechanisms for sequencing, conditionalization, and iteration over a set of execution strands. Chapter 3 did not give a good sense, however, for how one invokes these mechanisms and manipulates type-level objects—indeed—*defines* the control and data flow constructs for the analysis of an exploratory software system. That is the subject of this chapter on the language, as opposed to the mechanisms, of type flow analysis. It discusses the type-level control and data flow definition language in Essie.

Essie does not implement any single type analysis system. Instead, Essie is a programming language with which a user builds a particular instance of a type analysis system[1]. Essie's architecture greatly facilitates experimentation with different forms of control and data flow, as well as supporting the incremental development of sophisticated type analysis systems. This chapter

[1]This dissertation reports on only one of the type analysis systems built with Essie. Therefore, references to "Essie" may mean either "the Essie programming language" or "the type analysis system built with Essie for this dissertation." When deemed necessary, these alternative word senses (acronym senses ?) will be disambiguated.

presents only a small portion of the language constructs in ESSIE[2], which suffices to illustrate the general approach to defining an instance of a type flow analysis system. This chapter does not attempt to describe the language of type flow analysis in sufficient detail and completeness to enable construction of actual type analysis systems, which would be the province of a user guide.

This chapter presents two major levels of ESSIE's programming language. Sections 4.1 and 4.2 concern the first level: a set of *declaration* and *invocation* facilities. The declaration mechanisms allow an end-user to define the type-level structure of global variables and user-defined types implemented through the Lisp `defstruct`[3] mechanism. These declaration facilities require no specialized knowledge about the mechanisms of type flow analysis. The invocation mechanisms provide the top-level interface to the type flow analysis mechanisms. Along with various display facilities, this level includes all that day-to-day users of ESSIE manipulate.

Actual analysis of exploratory software using the first level of ESSIE consists of the following process. First, both ESSIE and the system to be analyzed are loaded into memory. Next, the developer declares the type-level structure of all global symbols and data structures using ESSIE's declaration constructs. The first part of Appendix B shows these global declarations for the system PTI. Next, the developer uses the invocation mechanisms of ESSIE

[2]As a rough indication of size, the current implementation of ESSIE contains approximately 600 functions.

[3]In this chapter, ESSIE language constructs are written in **boldface** font, while other functions are written in `typewriter` font.

to manually[4] perform a bottom-up traversal of the source code of the system to be analyzed, which incrementally builds the type-level structural representation of the system. During this analysis, problematic language constructs (such as `catch` and `throw`) or structural features (such as recursion without a type-level fixed point) may be encountered. These problems are circumvented through further use of declaration mechanisms to annotate their structural features. The second part of Appendix B shows examples of these declarations. For each function, ESSIE will either successfully produce a type level representation (as illustrated in Appendix A), or else produce an error message indicating the problem. This process of bottom-up traversal, analysis, and possible annotation continues until the entire system has been analyzed.

The second level of ESSIE also has two parts: one for data flow definition and one for control flow definition. The data flow definition facility defines the spectrum of type-level objects manipulated during analysis, such as integers, lists, hash-tables, and so forth. Approximately 16 such type-level objects are defined in the type analysis system currently used in this research. Defining a new type level object (termed an *analysis object*) requires providing its name, its internal type-level structure (if any), as well as a set of functions that implement properties required of all analysis objects. For example, each analysis object definition must supply a function that implements unification of two instances of the objects with each other. Section 4.3 describes these data flow language constructs and provides examples of their use.

[4]Automated traversal mechanisms are straightforward, but currently unimplemented, extensions to ESSIE.

The other part of the second level of ESSIE provides constructs for control flow definition. These constructs define the type-level interpretation of Lisp control flow forms such as **defun**, **let**, **cond**, and **progn**. This part of ESSIE is also used to bootstrap a type analysis system with interpretations for primitive functions such as **car**, **cdr**, **+**, **gethash**, and so forth. Around 80 such analyzers are defined in the current implementation of ESSIE.

Defining a control flow structure consists of describing, given the Lisp form and the braid representing the set of execution paths up to the occurrence of this form, the braid resulting from the type flow through the form. Section 4.4 describes control flow definition constructs along with definitions of their use.

The second level for control and data flow definition is not accessible to type flow novices: unlike the declaration facilities, control and data flow definition requires relatively complete understanding of type flow analysis and of the remaining levels in the ESSIE system. However, much of the information specified at this level, such as the control flow of forms like **progn** is generally domain-independent. Appendix D provides examples of analysis and analyzer objects found in the current implementation of ESSIE.

Beneath the second level lie the facilities for braid and strand manipulation, as well as the unification mechanisms. None of the facilities at this third level will be described in this chapter. Mechanisms for braid and execution strand manipulation were presented in Chapter 3, and the unification mechanisms are currently straightforward implementations of well-known algorithms

[Kni89]. In Section 6.3.1, some interesting extensions to the third level and their impact on higher levels of the system will be outlined. This chapter, however, now turns to the first language level of ESSIE, and its mechanisms for type declaration.

4.1 Declaration language facilities

Polymorphic inference of the type of a global *variable* (or global data structure) is problematic because, unlike functions, variables don't have a "definition" from which to derive a representation of their polymorphism. Instead, inferring polymorphism of variables requires examining every context in which each variable appears throughout the program. Once one context adds an additional polymorphic interpretation to a variable, every prior context must be reanalyzed in light of this new information. Although such an iterative approach could be implemented, it would incur tremendous computational overhead for systems of nontrivial size with nontrivial numbers of global objects. For this reason, ESSIE supplies a declaration language for global variables and data structures. This language has two top-level interfaces: **define-symbol-analysis** for variables and **define-defstruct-analysis** for structures.

4.1.1 Define-symbol-analysis

Lisp allows users to define global variables that can be referenced anywhere in the program, and that return the most recent value stored there. ESSIE

allows users to declare the type or types of these values using **define-symbol-analysis**.

Define-symbol-analysis *name* **&key** *constraints* [*Macro*]

Declares the type of the global variable *name* from the description *constraints*.

Define-symbol-analysis results in defining *name* to be a function of no arguments that returns a braid. The set of execution strands constituting this braid correspond to the different potential types that the variable could take on as expressed through the *constraints* argument. The *constraints* argument is a logical expression with terms consisting of type predicates and predicate arguments referring to *name*. For example, the following call to **define-symbol-analysis** declares `*table*` to be a hash table whose keys are of type symbol (inferred from the use of `'key`) and whose stored values are either lists or strings:

```
(define-symbol-analysis *table*
  :constraints (and (hash-table-p *table*)
                    (or (listp (gethash 'key *table*))
                        (stringp (gethash 'key *table*)))))
```

`*Table*` is an example of a heterogeneous data object: the values in the table can be either lists or strings. The ability to represent heterogeneous data objects is a novel aspect of type flow analysis, and results from a synergy between its control and data flow representations. Heterogeneity is discussed further in Section 4.6 below.

As the above example illustrates, declaring the type-level structure of symbols is relatively straightforward in ESSIE. Declaring Lisp **structure** types, as described below, presents more interesting and subtle issues that more fully exploit the language of type flow analysis.

4.1.2 Define-defstruct-analysis

Lisp provides the **defstruct** facility for defining structured objects analogous to the record type in Pascal. Consider the following call to **defstruct**:

```
(defstruct tree
  node
  left-child
  right-child)
```

This form results in the definition of a structured object named tree, as well as the definition of a set of functions to manipulate the object: `make-tree` returns an instance of a tree, `tree-node` returns the value of the `node` slot, `tree-left-child` returns the value of the `left-child` slot, and so forth. In addition, a type predicate function, `tree-p` is also automatically defined to allow testing for trees in the same way that predefined types such as integers and strings are tested for.

ESSIE provides **define-defstruct-analysis** to declare the type structure of Lisp structures.

Define-defstruct-analysis *name* **&key** *conc-name constructor constraints include* [*Macro*]

Declares the type level structure of the structure *name* as well as of its internal slots. If the defstruct definition of *name* supplies the *conc-name*, *con-*

structor, or *include* options, they must be supplied to **define-defstruct-analysis** as well. The *constraints* argument provides type-level constraints on the slots in instances of *name*.

Define-defstruct-analysis is employed in much the same way as the variable declaration facility **define-symbol-analysis**. Reference to slot names in the arguments to type predicates constrains the type of values that can be stored there. In addition, the structure name itself or any other structure name can be referenced in the constraint specification, which enables the declaration of recursive and mutually recursive data structures[5]. The following shows a declaration for the tree structure:

```
(define-defstruct-analysis tree
  :constraints (and (integerp node)
                    (or (null left-child)
                        (tree-p left-child))
                    (or (null right-child)
                        (tree-p right-child))))
```

This declaration indicates that the node slot contains an integer, and that the left and right child slots contain either `nil` or another instance of a tree structure.

This call to **define-defstruct-analysis** results in the generation of a lot of structural information:

[5]Supporting mutual recursion complicates **define-defstruct-analysis**, since completing the analysis of the constraint specification requires information from the as-yet unmade invocations of **define-defstruct-analysis** on the other recursive structures. ESSIE solves the problem through a form of "delayed evaluation"—the declaration process is procedurally encoded and completed during the first request by ESSIE for the analysis information. Since declarations must precede the actual analysis of the system, the missing information is guaranteed to be available by that time.

c. A new type-level analysis object in the system (through a call to **define-analysis**—see Section 4.3.1).

- The type level structure for the constructor function `make-tree`—a braid that accepts optional arguments for the slot values, ensures that they are of the required type, and returns a tree object as the result.

- The type level structure for the slot accessor functions `tree-node`, `tree-left-child`, and `tree-right-child`. These functions each accept a tree object as their single argument. `Tree-node` always returns an integer, while the left and right children return two strands, one with a result type of `nil` and one with a result type of a `tree` object.

- The declaration of `tree-p` as a valid type predicate function through an invocation of **define-type-predicate**—see Section 4.4.3.

Structure definitions result in the same capabilities for and limitations on heterogeneous data structures as illustrated in the declaration of `*table*`. Section 5.2.3 discusses a particularly interesting situation involving heterogeneous structures, which provides insight into some of the limitations of type flow analysis.

The *include* option to **define-defstruct-analysis** can be used to define type hierarchies, just as the `include` option to `defstruct` is used to build user-defined type hierarchies. Supplying the *include* option to **define-defstruct-analysis** results in ESSIE increasing the set of acceptable objects to the included slot accessor functions. For example, one way to define a tree that holds two values at each point is by defining a new structure that includes the tree structure as follows:

```
(defstruct (tree-graft (:include tree))
  value)
```

This call defines the constructor `make-tree-graft`, as well as slot accessors `tree-graft-node`, `tree-graft-value`, `tree-graft-left-child`, and `tree-graft-right-child`. In other words, tree-graft inherits the structure of tree. What's more, the slot accessors for tree can now accept instances of tree-graft objects. The corresponding invocation of **define-defstruct-analysis** for tree-graft results in the redefinition of the slot accessors `tree-left-child`, `tree-right-child`, and `tree-node` to accept not only instances of trees, but instances of tree-grafts as well. The type-level constraints on the slots shared between the two objects are inherited as well.

4.1.3 Define-fn-analysis

In a mature implementation of type flow analysis that handles all the control and data flow in a Lisp system, **define-symbol-analysis** and **define-defstruct-analysis** might be the only declaration constructs needed by users. In its current prototypical form, however, ESSIE must sometimes be "helped over the rough spots" in places where unimplemented forms of control flow (such as `return-from`, `catch`, and `throw`) or data flow (such as heterogeneous uses of `list`) are encountered. **Define-fn-analysis** provides one kind of aid for these situations.

Define-fn-analysis *name fn-form* [*Macro*]

Defines *fn-form* to be a type-level structural synonym for the function *name*. *Fn-form* must be a call to `function` with a lambda form as its

argument, where the parameter list in *fn-form* is identical to the parameter list in *name*.

Define-fn-analysis allows ESSIE to perform a structural analysis of *fn-form* instead of a structural analysis of the function definition associated with *name*. *Fn-form* should be homomorphic on the type-level to the structure of *name*. In other words, the braid resulting from *fn-form* should be equivalent to the braid for the function corresponding to *name*.

For example, consider a function `traverse-tree-hack` that travels around an instance of the tree structure defined previously, utilizing `return-from`, `catch`, and `throw` in order to return a particular node value in a really complex and clever way. Unfortunately, this function is too clever for ESSIE, and so the developer must either restructure the code into an analyzable form, or else tell ESSIE through **define-fn-analysis** what it's missing. The choice is obvious:

```
(define-fn-analysis traverse-tree-hack
  #'(lambda (tree)
              (tree-value tree))))
```

This call tells ESSIE that `traverse-tree-hack` is a function of one argument called `tree`, where `tree` must be an instance of a tree structure and the value returned is an integer.

4.1.4 Define-form-in-fn

Sometimes ESSIE only needs a little bit of help during analysis: perhaps only a single expression in the function proves problematic. Rather than assert the structure of the entire function in these cases, and lose the type assessment

facilities of ESSIE entirely, the developer may employ **define-form-in-fn** to assert the type level results of a single form.

Define-form-in-fn *form fn-name fn-form* [*Macro*]

Defines the type level effects of executing *form* within the function definition of *fn-name* as if *fn-name* were, in this case only, equivalent to the definition provided by *fn-form*. *Fn-form* must be a call to `function` with a lambda form as its argument, where the parameter list in *fn-form* is identical to the parameter list in *fn-name*.

Define-form-in-fn corresponds to a "local" version of **define-fn-analysis**. For example, the following declares that a particular call to `catch` within the function `traverse-tree-hack` will return an integer:

```
(define-form-in-fn  (catch 'node (gnarly-traverse node))
                    traverse-tree-hack
                    #'(lambda (tag &body body)
                        1))
```

Define-symbol-analysis, **define-defstruct-analysis**, **define-fn-analysis**, and **define-form-in-fn** implement a basic declaration facility for ESSIE. For use in real settings, however, some form of declaration monitoring and report facility must be added to help the user know what part of the system analysis depends upon what declarations, and conversely, the impact of a particular declaration on the rest of the system. ESSIE currently does not distinguish between function types that have been ascertained through analysis and function types that have been asserted through declarations. Adding this distinction forms a future implementation goal.

4.2 Invocation language facilities

Having declared the type structure of global symbols and data structures, the next step is to actually analyze the functions in the system under study. ESSIE provides two interfaces to the actual type flow analysis mechanisms: **analyze-expression** and **analyze-mutually-recursive-fns**. This section describes them as well as the type structure pretty printing feature, **display-fn-type**, which spares the user from needing to deduce the general type level properties of a function from a complex braid structure.

4.2.1 Analyze-expression

Analyze-expression is the fundamental interface to type flow analysis.

Analyze-expression *form* `&optional` *braid-or-strand* [*Function*]

Accepts a *form*, and an optional *braid-or-strand*, and returns a new braid. This braid represents the type flow analysis through `form` given the one or more execution strand contexts provided in *braid-or-strand*. *Braid-or-strand* defaults to an initialized braid containing a single strand with a null feasibility condition, an empty constraint environment, and no result type.

Figure 25 illustrates two simple invocations of **analyze-expression**; Chapter 5 and the appendices illustrate more complex usages from the analysis of a small software system.

In the first example, a call to `mapcar` is analyzed. This call involves a function and two lists as arguments, so **analyze-expression** is recursively

invoked to return the braids for the function and for the lists. The functional argument's type structure is then applied to the element types of the two lists, and a list of the result of the application results. Just as evaluating this form in Lisp at the value level returns `(30 50 80)`, the type flow analysis of this form returns `LIST[Int]`.

The second example shows the results of analyzing the recursive function **mem**, which implements an operation similar to **member**. The result type of this analysis is a function structure, which is itself a braid. Analysis of **defun** forms has the side-effect of storing this analysis in a global environment structure where it can be recalled for type flow application when the function is used.

Since **mem** is defined without type predicates, there is no feasibility condition. However, the constraint environment of the functional braid does contain the bindings for the parameters to the function, which are needed when the function **mem** is later applied to arguments. Chapter 3 described the process of analyzing functions to generate their braid structure in detail.

Typing **analyze-expression** and quoting function definition forms gets tiresome quickly, and so the current implementation of ESSIE includes a simple editor-based interface: positioning the cursor at a beginning of a function definition in a file and pressing SUPER-T invokes **analyze-expression**.

4.2.2 Analyze-mutually-recursive-fns

Analyze-expression suffices for expressions and singly recursive functions, but as described in Chapter 3, analysis of multiple recursion requires the set of recursive function definitions to be analyzed together as a group.

• *Invoking* analyze-expression *with a simple function application:*

```
(analyze-expression '(mapcar #'(lambda (x y)
                                 (+ x y))
                           '(20 30 50) '(10 20 30)))

===> Br[
       FC: NIL
       CE: <Empty>
       RT: LIST[ Int]
       ]
```

• *Invoking* analyze-expression *with a simple function definition:*

```
(analyze-expression '(defun mem (i lst)
                      "Like the Lisp Member function? :-)"
                      (if (equal i (car lst))
                          i
                          (mem i (cdr lst)))))
==> Br[
      FC: NIL
      CE: <Empty>
      RT: {Fn  Br[
                 FC: NIL
                 CE: < (I TV-303)(LST LIST[ TV-304]) >
                 RT: TV-303
           ]}
```

Figure 25: Two examples of ANALYZE-EXPRESSION

To support this, ESSIE provides a second interface to the type flow analysis mechanisms, **analyze-mutually-recursive-fns.**

Analyze-mutually-recursive-fns *fn-names* [*Function*]

Performs type flow analysis on a set of mutually recursive functions. **Analyze-mutually-recursive-fns** is passed a list of function names. It retrieves their definition, analyzes their recursive structure, and stores the analysis of all of the functions in the global environment. **Analyze-mutually-recursive-fns** returns `nil`.

This function has little observable external behavior, and its internal behavior is documented in Chapter 3. What is important to note about this function is that it reveals another interface shortcoming of the current implementation of ESSIE: users must order the sequence of invocations of **analyze-expression** and **analyze-mutually-recursive-fns** by hand in order to build the analysis of a software system. Such a system-level analysis invocation tool is very straightforward to provide but is not yet implemented[6].

4.2.3 Display-fn-type

The representation of a function as a set of execution strands with accompaning feasibility conditions, constraint environments, and result type is needed to correctly infer the structural effects of the function at the type level. It is not, however, very useful as documentation of what users most often wish to know: what are acceptable argument and result types to a function? To

[6][Hec77] provides an algorithm for determining the sets of mutually recursive functions, or the "strongly connected components" from the static call graph of a system

satisfy this need, ESSIE includes a facility that retrieves the braid representation of a given function from the global environment, and converts it to a representation of its acceptable argument and result types, as illustrated in Figure 26.

Display-fn-type *fn-name* [*Function*]

Prints a representation of the set of argument and result types. **Display-fn-type** takes the name of an analyzed function *fn-name*, retrieves its braid, and processes it into a more readable form.

In producing the simpler representation, **display-fn-type** loses some information—that of the feasibility condition. While this information is crucial to distinguishing between infeasible execution strands and type-level errors, removing it allows a succinct and readable representation of the set of argument/result types supported by the function.

While `if-p` is simple enough that the braid is relatively readable, the gains in perspicuity produced by **display-fn-analysis** can sometimes be considerable. For example, the function `unify-type` is listed in both formats in the Appendix. The "displayed" version requires 7 lines of output, from which it is easily seen that `unify-type` takes combinations of type variables and type operators as arguments, resulting in either a list or `nil`. The corresponding braid occupies 48 lines and over a page of output, from which it is hard to see anything at all about the structure of `unify-type`.

The declaration, invocation, and structural display facilities described above form the first level of ESSIE. The next two sections describe the second

• *The normal braid representation for* `if-p`*:*

```
(IF-P {Fn Br[
  FC: ((LISTP LIST[ TV-259]))
  CE: < (EXP LIST[ TV-259]) >
  RT: Boolean[T]

  FC: ((NOT (LISTP TV-282)))
  CE: < (EXP TV-282) >
  RT: Boolean[F]

  FC: ((LISTP LIST[ TV-372]))
  CE: < (EXP LIST[ TV-372]) >
  RT: Boolean[F]
  ]})
```

• *The "display" form of* `if-p`*:*

```
(IF-P
Function
 < (EXP LIST[ TV-3]) > ==>> Boolean[T]
 < (EXP TV-8) > ==>> Boolean[F]
   Where: (NOT (LISTP TV-8))
 < (EXP LIST[ TV-12]) > ==>> Boolean[F])
```

Figure 26: Two views of the type of IF-P

level of ESSIE, which provides constructs for defining the data and control flow for the system.

4.3 Data flow language facilities

One view of ESSIE is as a symbolic execution system at the type level—in other words, it takes a program consisting of functions that compute with and pass around values, and produces a type-level version: for each of the original functions, ESSIE produces a new function that computes with and passes around types. The data flow language allows the developer to define the kinds of types "passed around" in this type-level system. They are termed *analysis objects*, and the principal mechanism for their creation is **define-analysis**.

4.3.1 Define-analysis

Define-analysis defines new type-level objects that can be manipulated by ESSIE. Each type-level object in ESSIE must support a set of operations in order to integrate with the type flow analysis mechanisms. **Define-analysis** provides a way for the developer to specify both the internal structure of a type-level object, as well as the implementation of the set of required operations upon this object. By encapsulating data and procedure together in this manner, ESSIE's data flow language provides the abstraction benefits of object-orientation.

Define-analysis *name* **&key** *slots unifier equal-p copier occurs-checker component-p constructor print-function* [*Macro*]

Specifies the structure and implementation of an analysis object. It returns the symbol *name*.

The function and properties of each argument to **define-analysis** are discussed in the following sections. To give an overall sense for the use of **define-analysis**, an example definition for the Lisp hash-table type is provided in Figure 27.

Specifying internal structure with slots

Some analysis objects, such as integers and strings, are atomic—they have no internal structure. Other objects, such as lists and hash tables, have internal structure. The *slots* argument to **define-analysis** takes a list of *slot-names*, each of which becomes an internal accessible structure in the analysis object. In the hash table example in Figure 27, slots are provided for the two internal components of a hash table: the keys and their associated values. The resulting slots, `table.key` and `table.val`, can be referred to in subsequent arguments of the analysis object definition.

Specifying the unification procedure with unifier

A fundamental operation in type flow analysis is unification. From an implementation standpoint, unification is the process of making two analysis objects "logically identical" with respect to the rest of the system, or

- *The analysis definition for a hash table:*

```
(define-analysis table
  :slots (key val)

  :unifier
  #'(lambda (analysis1 analysis2)
      (cond ((and (table-p analysis1) (table-p analysis2))
             (let ((key-subst (unify-analysis (table.key analysis1)
                                              (table.key analysis2)))
                   (val-subst (unify-analysis (table.val analysis1)
                                              (table.val analysis2))))
               (when (and (subst.success key-subst)
                          (subst.success val-subst))
                 (make-subst :success t
                             :list (append (subst.list key-subst)
                                           (subst.list val-subst))))))
            (t ;;analysis1 or analysis2 is not a hash table.
             (make-subst :success nil))))

  :copier
  #'(lambda (table env)
      (make-table :key (fresh-analysis (table.key table) env)
                  :value (fresh-analysis (table.val table) env)))

  :equal-p
   #'(lambda (tab1 tab2)
       (and (equal-analyses-p (table.key tab1) (table.key tab2))
            (equal-analyses-p (table.val tab1) (table.val tab2))))

  :occurs-checker
  #'(lambda (table type-var)
      (or (occurs-in-analysis-p type-var (table.key table))
          (occurs-in-analysis-p type-var (table.val table)))))
```

Figure 27: A definition of a hash table analysis object

else determining that this cannot be accomplished. The *unifier* argument to **define-analysis** specifies how this is done.

The unifier argument is required to be a function that accepts two arguments, which could be instances of any currently defined analysis object in the system. The function returns a *substitution structure*, which is an object that indicates whether or not unification succeeded, and if so, what type variable instantiations must be performed in order to accomplish it. Conceptually, the unifier argument tests to see that the passed arguments are instances of the analysis object under definition, and then "walks" the internal structure, checking to see that the corresponding internal components can be unified. The unifier argument in Figure 27 provides an example of this process.

To simplify the recursive walking of analysis objects during unification, ESSIE provides a generic interface to the set of all *unifier* functions called **unify-analysis**. This function accepts two analysis objects, and then attempts to unify them by successively invoking all of the currently defined *unifier* functions until one succeeds. As illustrated by the hash table analysis definition, **unify-analysis** allows *unifier* functions to be specified without knowledge of what other analysis objects exist in the system.

Specifying the generation of structural copies

The process of type flow analysis frequently involves making structural copies of braids and execution strands. This copying process ultimately involves copying individual analysis objects. The *copier* argument to **define-analysis** takes an instance of the analysis object under definition, plus an envi-

ronment structure of non-generic variables[7], and returns a new structural copy of the analysis object. "Structural" copy means that all of the type variables in the analysis object have been replaced by new instances of type variables, and that co-ocurrences of the type variables are correctly maintained.

As in the case of the *unifier* function, the *copier* function traverses the internal structure of the analysis object and, again, ESSIE provides a generic function to allow all analysis objects to be defined independently of each other. This generic function is called **fresh-analysis**. It takes an analysis object and an environment, and dispatches to the correct instance of a *copier* function.

Determining analysis object equality

ESSIE frequently needs to know if two objects are structurally equal, for example, when attempting to filter duplicate execution strands from a braid, or within analyzers such as **setf** (discussed below). One way to do this is to attempt to unify the two objects (without actually performing the instantiations), and see if the resulting substitution list indicates only the instantiation of one type variable to another. If the only substitutions are type variable for type variable, then the two analysis objects are equal in a "structural" sense.

While this unification procedure works, it involves a lot of unecessary construction of substitution lists. ESSIE provides this unification-based strategy as a default *equal-p* function, but allows developers to supply a *equal-p* function that simply tests for object equality and returns `t` or `nil`, without creating

[7]Unfortunately, the **define-analysis** abstraction is not absolutely perfect: this environment structure needs to be passed in and through each *copier* function on the way to lower language levels of the implementation, but is normally not manipulated on this level.

substitution lists. The hash-table example in Figure 27 illustrates this more efficient implementation, as well as the by-now familiar provision of a generic function, in this case called **equal-analyses-p**, to simplify the testing of interior components.

Specifying the unification occurs check

In order for unification to be defined for arbitrary type expressions, it requires an "occurs check", i.e., a test to ensure that no circular instantiations of type variables are built. However, the occurs check also limits the kinds of things that can be unified, as well as incurring a great deal of computational overhead. For this reason, many unification-based systems (such as Prolog) omit the use of the occurs-check.

ESSIE allows developers to use or omit an occurs check as desired. The default *occurs-checker* function always returns `nil`, which has the effect of disabling the occurs-checking process. When supplied, the *occurs-checker* function must accept an instance of the analysis object under definition, as well as a type variable, and test to see if that type variable occurs anywhere in the internal structure of the analysis object. As usual, ESSIE supplies a generic function, **occurs-in-analysis-p**, to support this process. Figure 27 shows the occurs-check function for hash-tables.

Specifying analysis components, constructor, and print-functions

Several other optional arguments that may be provided in the definition of an analysis object are omitted from the example definition of hash-tables.

The *component-p* argument specifies a function that tests to see if an arbitrary analysis object occurs in an instance of the object under definition. This is used for various "housekeeping" operations in ESSIE such as the removal of vestigial feasibility condition terms.

The *constructor* argument allows developers to encapsulate the default constructor function within a user-defined constructor, so that additional computation or analysis can be performed during the creation of an object.

Finally, the *print-function* argument allows developers to provide a standard `defstruct`-style print function to improve the formatting of the output.

4.3.2 Define-primitive-analysis

As the preceeding discussion implies, many of the operations needed to integrate a new analysis object into the type flow analysis machinery involve the inspection of the internal structure of the analysis object under definition. However, many objects (integers, strings, and so forth) have no internal structure. For these objects, it is possible to entirely automate the construction of the functional arguments to **define-analysis**, and so ESSIE provides a specialized interface to **define-analysis** for these situations.

Define-primitive-analysis *name* **&key** *print-function* [*Macro*]

Defines a simple, slotless analysis object *name*. Results in a call to **define-analysis** with all functional arguments automatically generated. *Print-function* may be supplied to improve output formatting.

4.3.3 Issues in representing functional data objects

Define-analysis and **define-primitive-analysis** provide a natural mechanism for the definition of the traditional data types such as integers, strings, and hash tables. However, some of these operations present a challenge when applied to less traditional data types available in exploratory languages, such as ad hoc polymorphic functions. What does it mean to "unify" two such *functions*, for example? This issue had to be addressed in order to implement recursion, where a representation of the recursive function had to be maintained within each execution strand's constraint environment as an analysis object.

ESSIE solves this dilemma by representing functions in two ways. The "normal" representation is a braid, and this representation for functions cannot be unified with other objects. The second representation consists of a set of invocation structures that record the context in which the function was used. This representation can be unified with other instances of this representation, and the result is to simply share the combined set of invocation structures. In the case of recursion, this functional representation is eliminated during the recursive resolution phase of function definition processing.

4.4 Control flow language facilities

The final language facility of ESSIE to be described in this chapter concerns the definition of control flow, or how the analysis objects discussed in the last section are actually passed around in any particular instance of a type analysis

system. As illustrated in Appendix D, ESSIE provides **define-analyzer** as the principal interface to the control aspects of type flow analysis.

4.4.1 Define-analyzer

Define-analyzer specifies the control flow characteristics of a Lisp construct.

Define-analyzer *form braid-or-strand* &body *body* [*Macro*]

Defines how control flow passes through *form*, where *braid-or-strand* represents the one or more execution strands generated through analysis up to *form*. *Body* returns a braid representing the set of execution strands resulting from the type flow analysis of *form*.

Examples of currently implemented analyzers in ESSIE

Define-analyzer is used to build the type-level representation of control structures as well as bootstrap the system with predefined functions. ESSIE currently includes around 80 analyzers. Examples of the principal control flow analyzers include:

- **Progn:** The first form in the body of the progn is analyzed with respect to the set of execution strands in *braid-or-strand*. The braid resulting from this analysis is used as context for the analysis of the second form in the body of the progn, and so on. The braid resulting from the analysis of the last form in the progn is returned.

  Other sequential analyzers like **prog1** are simple variants of **progn**.

- **Let:** The let analyzer is responsible for augmenting the constraint environments of each of the incoming execution strands with new identifier names and their analyses. The body of the `let` form is then analyzed (using the **progn** analyzer) with these augmented constraint environments. The resulting braid is returned, after having popped the let-bound identifiers from the constraint environments.

  The **let*** analyzer is similar, with the difference that the type flow analysis of each identifier binding is available to the next. (Related, but as yet unimplemented analyzers for **flet** and **labels** will allow local functional bindings.)

- **Cond:** The **cond** analyzer returns the results of analysis of conditional forms. The techniques employed in this analyzer are covered in detail in Section 3.5.

  Other analyzers built on, or directly related to **cond** include: **if**, **when**, **unless**, and **case**.

- **Defun:** The **defun** analyzer results in a braid with a *function* analysis object as the result type of each execution strand. As a side-effect, the name of the function and its corresponding analysis object is stored in a global environment. The **defun** analyzer works by augmenting the constraint environment with bindings for each parameter of the function, and then analyzing the body of the defun with the **progn** analyzer. The resulting braid is saved (including the parameters in the constraint environments).

  The **function** analyzer is similar to **defun**, except it does not side effect the global environment.

- **Mapcar:** The **mapcar** analyzer results in a braid representing the list constructed from applying its functional argument to its list arguments. All the many flavors of mapping functions are analyzed in a manner analogous to **mapcar**.

- **Setf:** The **setf** analyzer implements side-effects in ESSIE. The current implementation of **setf** is somewhat restrictive: it raises a warning if the value and the place to be setf'd are not equivalent type expressions. This is required since ESSIE does not currently model the data flow through actual storage locations—in other words, if a variable can take on three different types of values, ESSIE does not model which particular type the variable is bound to at every point in the program.

This list represents many, but not all, of Lisp's control flow constructs. The current implementation of ESSIE does not attempt to model non-local control flow, which thus excludes constructs like: `block`, `return`, `do`, `tagbody`, `go`, `catch`, and `unwind-protect`. In addition, the current implementation represents only a single value return type, which excludes `multiple-value-bind` and their ilk. These unimplemented features will be discussed further below.

Define-analyzer is also used to bootstrap the system with primitive Lisp functions. Examples of these include:

- **List:** The **list** analyzer currently supports only homogeneous lists, although heterogeneous lists can be partially modelled by multiple execution strands, each of which containing a different homogeneous list containing one of the heterogeneous elements. Section 4.6 contains more details on this.

Other primitive list functions are also represented as analyzers, such as **car**, **cdr**, **cons**, **append**, and so forth.

- **+:** The basic numeric analyzers such as +, -, /, and so forth are implemented in ESSIE. However, only a single kind of numeric analysis object (integer) has been implemented in ESSIE, so these analyzers are monomorphic. Better representations for the various flavors of numeric objects (fixnums, bignums, floating point, etc.) and their relationships will result from more complete modelling of the Common Lisp type hierarchy, as discussed in Section 6.3.2.

- **And:** The logical connectives **and**, **or**, and **not** are implemented and model their short-circuiting semantics.

- **Predicates:** Varieties of predicates including the various flavors of **equal**, numeric predicates like <, and so forth have been implemented.

- **Miscellaneous:** Many other primitive Lisp functions have been defined during the evaluation of ESSIE, ranging from **make-hash-table** to **format** to **si:%pointer**. However, the current total of 80 analyzers is only 10% of the total number of primitive Lisp functions, so many more remain to be defined.

Having given a sense for the scope of currently implemented analyzers in ESSIE, the next section provides a brief introduction to the way in which analyzers are actually built.

An Analyzer for **Gethash**

Figure 28 illustrates an analyzer for the predefined Lisp function **gethash**. This analyzer works in concert with the **table** analysis object defined in the previous section to support the manipulation of hash table objects at the type level.

The **gethash** analyzer works in the following general way[8]. First, it analyzes the arguments to get their type-level representation. Next, it checks to make sure that the second argument to **gethash** can be constrained to a hash table. Next, it makes sure that the key argument to **gethash** can be constrained to the type of key of the hashtable object. If so, it returns a new braid whose result is the type of value stored in the hash table. Otherwise, an error message is signalled.

This simple description glosses over several issues. First, this analyzer can be called with either a single execution strand or a braid. How is this ad hoc polymorphism in the analyzer arguments accommodated? Second, whenever a Lisp expression is analyzed, a *braid* results, representing all the various polymorphic possibilities for its value. When the arguments in the **gethash** form are analyzed, each of them may result in multiple possibilities for their actual analysis value. To fully explore the possibilities requires testing each single execution strand and result type of one argument against each of the other single execution strands in all the other arguments. How is this situation accommodated?

[8]Note that this simplified version of the analyzer does not process the third argument to gethash (the default value to be returned if the key is not found), nor represent that multiple values are returned.

- *The analyzer for the gethash predefined function:*

```
(define-analyzer gethash (form braid-or-strand)
  "Analyzes gethash's args and returns the value of the table."

  (with-strand (strand braid-or-strand)
    (with-arg-analyses ((fc env args)
                        (analyze-exps (cdr form) strand))
      (let ((key-arg (first args))
            (table-arg (second args)))

        (unless (do-unification table-arg (make-table) nil)
          (essie-error "Gethash's arg not a hashtable: ~s" form))

        (let ((table-key (table.key (prune table-arg)))
              (table-value (table.value (prune table-arg))))

          (cond ((do-unification key-arg table-key nil)
                 (make-singleton-braid :fc fc
                                       :env env
                                       :result table-value))

                (t
                 (essie-error "Hashtable key inconsistent: ~s"
                              form))))))))
```

Figure 28: The analyzer for GETHASH

The answer to the first question is **with-strand** and the answer to the second question is **with-arg-analyses**. Together they abstract away virtually all of the multiple strand and multiple argument analysis type overhead from the writing of analyzers. The next section describes them briefly.

4.4.2 With-strand and With-arg-analyses

With-strand *strand braid-or-strand* &body *body* [*Macro*]

If *braid-or-strand* is a single strand, then **with-strand** simply binds *strand* to that strand and executes *body* with that binding. The result returned from *body* must be a braid. If *braid-or-strand* is a braid, then **with-strand** successively binds *strand* to each of the strands in *braid-or-strand*, and executes *body* once for each of those bindings. The braids resulting from each of those executions of *body* are then reprocessed into a single braid, which is returned.

With-strand is an extremely useful abstraction in ESSIE. It allows analyzers to be used in situations where the context of analysis consists of either a strand or a set of strands without any additional developmental overhead. With **with-strand**, the developer can uniformly implement the analyzer by referring to a single execution strand and returning a single braid.

With-arg-analyses *(fc env args) arg-braid-list* &body *body* [*Macro*]

Arg-braid-list is a set of braids, one for each argument being processed. **With-arg-analyses** separates each braid for each argument into its component set of strands, and generates the cross product representing all combinations of argument strand sets.

These sets are then consolidated down by combining their feasibility conditions together (eliminating from further consideration any consolidated conditions shown infeasible) and their constraint environments (signalling an error if a consolidated constraint environment is inconsistent after having obtained a feasible consolidated feasibility condition).

For each of the resulting consolidated execution strand sets, *body* is executed with *fc* bound to the consolidated feasibility condition, *env* bound to the consolidated constraint environment, and *args* bound to a list of argument analysis objects in the same order as their braids appeared in *arg-braid-list*.

For the reader armed with this information, the **gethash** analyzer should now be completely understandable. The call to **with-strand** takes care of the ambiguity surrounding *braid-or-strand*, so that further processing can simply refer to a single strand denoted by *strand*. In the next line of the analyzer definition, all of the arguments in the call to **gethash** are analyzed with **analyze-exps** to form a list of argument braids. These braids are processed with **with-arg-analyses**, so that the remainder of the analyzer can manipulate analysis objects like integer, string, and so forth. Next, the analysis objects bound to the key and value arguments are extracted from the arguments list, and the remaining processing simply tests to make sure they are of appropriate type.

This concludes the discussion of **define-analyzer**. The last control flow construct to be discussed is **define-type-predicate**, which defines the semantics of this special class of Lisp functions to ESSIE.

4.4.3 Define-type-predicate

Some of the type-level information gathered by ESSIE is generated in the same way as in other type inference systems: the types of identifiers are constrained by the contexts in which they are used. For example, in the expression `(+ 2 x)`, the identifier `x` can be inferred to be a number since + requires all of its arguments to be numbers.

Type flow analysis is unique among type inference systems in exploiting the presence of type predicate functions and the special way in which they reveal type information. First, type predicates are total functions, defined on every type of object, so that calling a type predicate with a particular object does not constrain the object's type in any way.

However, in combination with additional conditional test clause processing machinery, what type predicates *do* reveal is what type its argument *would have to be* in order for the conditional test to succeed, and the conditional arm to be executed.

Given an arbitrary argument to a type predicate, its corresponding analysis type could be one of three kinds. Consider, for concreteness, the `listp` type predicate and some argument `Y`. If the analysis of `Y` within the current strand indicates that `Y` is some sort of list[9], then the type predicate succeeds, and (given equal success with other predicates) the strand could be continued along this conditional arm.

Conversely, if `Y`'s analysis indicates that it is an integer, then the type

[9]Of course, the analysis of `Y` is always a *braid*, but with the help of with-strand and with-arg-analyses, this complexity can be abstracted away. I told you they were useful.

predicate is guaranteed to fail, and the execution strand cannot be continued along into the body of the conditional arm.

There is also a third possibility: that Y's analysis is *ambiguous* with respect to the success or failure of the conditional test. For example, if the current analysis of Y does not constrain it at all (i.e., Y's analysis is a type variable), then ESSIE must represent both success and failure, since later information could constrain Y to either a list or an integer.

ESSIE sorts out these different possibilities automatically using the information provided by **define-type-predicate**.

Define-type-predicate *name* **&key** *predicate-analysis fail-p* [*Macro*]

Specifies the type flow effect of *name* during the processing of conditional test forms. *Predicate-analysis* is an analysis object that will pass the type predicate. *Fail-p* is a function of one argument that, when passed an analysis object, returns T if the object could possibly fail the type predicate.

Figure 29 illustrates the type predicate definition for **symbolp**. Invocations of **define-type-predicate** expand into a call to **define-analyzer** that uses the *predicate-analysis* and *fail-p* arguments to inspect the argument to the type predicate. If the argument unifies against *predicate-analysis* and succeeds against *fail-p*, then the analyzer returns a braid with a single strand with a result type of boolean true. If the argument unifies against *predicate-analysis* and fails *fail-p*, then the analyzer returns a braid with two strands—one with a return type of boolean true and the other boolean false. Finally, if the argument does not unify against *predicate-analysis*, the analyzer returns a braid with a single boolean false strand.

• *The type predicate definition for* symbolp*:*

```
(define-type-predicate symbolp

  :predicate-analysis
  (make-symbol-analysis)

  :fail-p
   #'(lambda (analysis)
       (arg-check analysis analysis-p)
       (not (symbol-analysis-p analysis))))
```

Figure 29: The type predicate definition of SYMBOLP

This concludes the discussion of the principal control and data flow language features of ESSIE. Example uses of these constructs appear in Appendix D. The next section discusses some features not yet supported in the current implementation of type flow analysis.

4.5 Unsupported Lisp language features

The set of analysis and analyzer objects in the current type analysis system implement only a subset of the data objects and predefined functions in Common Lisp. However, producing a type analysis system for Common Lisp requires more than simply supplying the missing analysis and analyzer extensions: some representational extensions will be required as well. This section discusses some of specific extensions required to handle certain "advanced" language features of Common Lisp.

4.5.1 Dynamic function shadowing

In Common Lisp, it is possible to introduce new lexically scoped function definitions via `flet` and `labels`. These two constructs have an effect similar to `let`, except that they result in a local definition for the function cell, rather than the value cell of an identifier. ESSIE currently implements only a single cell (as in Scheme) rather than a separate function and value cell. With this minor extension, such *lexically* scoped function shadowing can be supported.

However, it is also possible to *dynamically*, as opposed to lexically, shadow function definitions. Static analysis techniques such as those in ESSIE cannot precisely characterize the scope and extent of dynamic function shadowing, which allows a function name such as `car` to be bound to different definitions depending upon the characteristics of the call stack at the time of the invocation of `car`. This capability is rarely used and often discouraged, and ESSIE cannot support this behavior. Extending ESSIE to (partially) support this behavior could consist of an additional declaration mechanism, allowing the specification of a *set* of function definitions to be associated with an identifier within a particular function.

4.5.2 The compilation environment

Common Lisp provides the `defmacro` facility, which allows source-level to source-level transformations and the ability to circumvent the normal evaluation mechanism for function arguments. Macros are expanded at compile-time, and may result in the definition and use of new data and functional objects while producing the transformed source-level representation. Since only the transformed source code is actually compiled into object code and stored, these

supporting data and functions are said to exist only in the "compile-time" environment.

ESSIE does not analyze macros directly, but rather analyzes the resulting source-level transformation. This is consonant with its mission of uncovering type-level errors in the run-time environment. However, it does mean that type-level errors can exist undetected by ESSIE in the compile-time environment. Support for the analysis of the compile-time environment represents a relatively straightforward extension to ESSIE.

4.5.3 Multiple values

Common Lisp allows functions to return more than one value at a time through the `values` construct, and various constructs such as `multiple-value-bind` and `multiple-value-setq` to receive the set of values returned from a multiple valued function.

ESSIE cannot currently represent multiple values for the simple reason that the result type object in execution strands only stores one type expression representing one return value. Much of the functionality of multiple values can be obtained by simply loosening this restriction. However, certain constructs such as `multiple-value-list` return the values as a list, which introduces the problem of heterogeneity as discussed below.

4.5.4 Function closures

Closures are a commonly supported feature of functional as well as exploratory languages. When a function represented as a lambda expression

is evaluated, a "functional closure" is returned, which consists of the function definition as well an environment that represents any lexically scoped bindings in effect at the time of the definition. For example:

```
(let ((x 5))
  #'(lambda (y)
      (+ x y)))
```

This `let` form returns a functional object as its value, where the functional object can refer to the binding of `x`.

ESSIE does not currently represent this ability of lambda expressions to close over lexically scoped identifier bindings. Adding the ability simply involves modelling the lexical closure facility of the Common Lisp compiler itself, which must identify whether or not any lexically scoped identifiers are referred to in the lambda expression, and then create a special environment structure to preserve this binding.

4.5.5 Type Declarations

Common Lisp includes the ability to declare the type of object returned by an expression through the declaration called `the`. The `the` form consists of an argument specifying a type and a form to be evaluated. For example:

```
(the integer (+ x 3))
```

`The` returns the value of its second argument, and it is an error for this value to not be of the type specified by its first argument. However, Common Lisp does not require implementations to *enforce* this type declaration. Instead, it is viewed as advice to the compiler that supports optimization of object code.

For obvious reasons, ESSIE ignores the type information declared by the forms. However, an interesting application of the type information derived by ESSIE is the *insertion* of the forms where possible in the source code. This future direction will be discussed in Section 6.3.3.

4.6 Representing heterogeneous data objects

Heterogeneous data objects are those whose structural components can contain a set of elements of more than one type. For example, a single Lisp list can contain an ordered sequence of integers, strings, hash-tables, functions, or any other data object.

Unfortunately, the type expression languages developed for parametric polymorphism are based upon the assumption of homogeneity: that composite data objects such as lists can hold only objects of one type at a time. These type expression languages can represent a list containing only integers (list[integer]), or a list containing only strings (list[string]), or even the set of all lists containing single types of objects (list[v_1]), but cannot express a list containing an integer *and* a string at the same time, or recursive list structure. By maintaining this restriction to homogeneity, parametric polymorphic languages can ensure that the type of object returned from loops is invariant over the number of iterations, and that standard forms of unification can be employed to collect and maintain constraints.

ESSIE's current data flow language implements a type expression language sharing the homogeneous restriction of the parametric polymorphic languages. However, this does not restrict ESSIE as a whole from the ability to represent

heterogeneity. This capability comes from the ability to *declare* (rather than infer) the type structure of objects as a set of execution paths.

For example, Figure 1 in Chapter 1 illustrated the workings of type flow analysis through an implementation of a stack object that could be implemented as either a list or a hashtable. In that discussion, it was assumed that any instance of a stack could contain objects of only a single type. However, this homogeneous restriction can be overcome through structural annotations.

For example, suppose that stacks can actually contain two different kinds of objects, such as strings and integers. ESSIE cannot infer such a type-level structure, since its type expression language can infer only homogeneous types. However, the developer can annotate the structure of the implementation of the stack data type to represent such heterogeneity, as shown in Figure 30.

Figure 30 annotates the function `push-list` by declaring the result from `cons`, which is normally interpreted by the type analysis system to be a heterogeneous function. In this case, the annotation extends the interpretation of `cons` to return two kinds of *homogeneous* lists: one containing integers and one containing strings. As a result, all users of the `push` function must accomodate stacks with either strings or integers, since any operation (such as `pop`) will return both a string and an integer as a possible return type.

This result approximates the type-level representation of a "true" heterogeneous list. Its representational shortcoming is that no ordering information is preserved: for example, it cannot represent a stack containing *alternating* integers and strings, such that successive calls to `pop` would alternate between returning an integer and a string. In this case, every call to `pop` would return

---

• *A simple definition of* `push-list` *from the stack type in Chapter 1:*

```
(defun push-list (item stack)
  (cons item stack))
```

• *This normally homogeneous use of* `cons` *can be made heterogeneous through annotations:*

```
(define-form-in-fn
  (cons item stack)
  push-list
  #'(lambda (obj1 obj2)
      (cond ((or (and (stringp obj1) (listp obj2))
                 (and (integerp obj1) (listp obj2)))
             (or (list "")
                 (list 1)))))
  )
```

Figure 30: Representing heterogeneity through annotations

two strands, one representing the integer possibility and one containing the string possibility.

If such an approximation of list structure is inadequate, the alternative under the current implementation of type flow analysis is restructuring into a set of globally declared structures, where such recursive structure can be expressed. Section 6.3.1 discusses future proposed research on "fuzzy unification," where the *inference* of such heterogeneous data objects will be developed.

4.7 Summary

This chapter presented an overview of the language facilities in type flow analysis. These constructs were organized into a set of layers according to the depth of knowledge about the mechanisms of type flow analysis required to exploit them. The first layer of declaration and invocation mechanisms requires relatively little knowledge about type flow analysis concepts. For example, writing a global symbol declaration simply requires knowing how to construct a Lisp boolean expression referring to type predicates. This layer corresponds to the "end-user" interface. While this layer is developed sufficiently to evaluate the current status of type flow analysis, additional facilities (such as an **analyze-system** function) must be provided in a releasable version of ESSIE.

The second layer of the system defines the control and data flow mechanisms, which result in the implementation of a particular instance of a type flow analysis system. Defining control and data flow constructs requires, in most cases, a relatively deep knowledge of type flow analysis. Furthermore,

individual control and data flow components are sometimes interdependent. For example, the **gethash** analyzer illustrated in Figure 28 refers to the **table** analysis object illustrated in Figure 27. While direct interdependencies of this nature are unavoidable, the control and data flow facilities do not require any "global" knowledge of the other analysis objects or analyzers currently defined in the system.[10]

The language of type flow analysis has evolved a great deal since the initial design of the system, and has now reached a sort of plateau: the current abstractions appear stable and appropriate for ESSIE's capabilities. In its structure, ESSIE exhibits many of the qualities associated with the advantages of exploratory development and languages.

First, ESSIE does not implement any particular type analysis system, but instead defines a language with which a developer can implement and experiment with a broad class of type analysis systems.

Second, this language provides both data abstractions (with the various **define-** constructs) and control abstractions (with the various **with-** constructs).

Third, extensions to the language are now defined in terms of the language itself: for example, the **define-defstruct-analysis** construct expands into calls to **define-primitive-analysis** and **define-type-predicate**, which then expand into calls to **define-analysis** and **define-analyzer**, and so forth.

The current plateau does not appear to be the summit, however. For example, future research on heterogeneous data objects in type flow analysis will

[10]Examples of which appear in Appendix D.

include exploration of "fuzzy unification," where the unification of two analysis objects may result in a set of possible results, including both success and failure. This major change to the third language layer of ESSIE will certainly make a significant impact on the definition of analysis objects and analyzers at the second layer, forcing the design of new facilities and the potential redesign of the current ones. Greater support for type hierarchies and value-level processing will also catalyze changes.

These interesting issues will be discussed in more detail in the final two chapters. The next chapter presents an evalution of type flow analysis and ESSIE as they stand now, and illustrates their strengths and weaknesses with the results from the type flow analysis of a small scale exploratory system, as well as a coarser type analysis of a major, large scale exploratory system. The final chapter charts the course ahead.

CHAPTER 5

EVALUATION

This chapter presents an evaluation of type flow analysis for exploratory software development. This evaluation explores the following issues. First, does type flow analysis address a recognized problem in software development? Second, what are the limitations of type flow analysis and of its current implementation? Which limitations result from the prototypical nature of the current implementation, and which result from deeper limitations of the theory?

The first section of this chapter provides evidence that type flow analysis addresses a recognized problem in software development through the review of a study [Wol89] comparing the implementation of two similar systems, one in a strongly typed language and the other in a weakly typed language. The conclusions reached in the study provide direct evidence that the approach of type flow analysis to combining the strengths of weakly and strongly typed software addresses a recognized, significant problem in software engineering.

The second section presents an evaluation of ESSIE, the current implementation of type flow analysis. This evaluation consists of the results of analysis of a small scale exploratory system called PTI. The results of this analysis reveal that ESSIE can accomodate many of the features of exploratory software,

can discover type errors, and can support the evolution of software structure toward a more robust form. However, the results also reveal many significant and interesting limitations of the current state of type flow analysis, and suggest future research directions.

Since ESSIE is not sufficiently developed to handle "real world" exploratory software, an alternative type level study was undertaken to see if a large scale exploratory software system contained the structural characteristics, specifically ad-hoc polymorphism, addressed by type flow analysis. The analysis also investigated whether structural constructs not currently supported by ESSIE, such as heterogeneous structures, form a significant component of the data objects used in large scale software. The results show that ad hoc polymorphism is a major software structuring technique in this instance of real world software, but that heterogeneous structures, albeit with small numbers of internal element types, do form a significant proportion of the data objects manipulated. Along with other data, this indicates that type flow analysis contains a sound representational basis, but will still require future research in order for it to scale up to larger software.

5.1 Literature-based evidence for type flow analysis

The most general evaluative question to ask of this research is simply, "Is there any evidence that attempting to combine the qualities of weakly and strongly typed languages in this manner solves a useful problem in software development?" Supporting evidence that type flow analysis does address a significant issue comes from [Wol89]—a study by Wolf comparing the use of

a strongly typed and weakly typed language for software development. This study compared two similar systems for describing and generating electronic hardware components. One of the systems was implemented in Flavors—a weakly typed, object oriented extension to Lisp, while the other was implemented in C++, a strongly typed extension to C. According to the author, the type-level distinction outweighed everything else: "Of all the differences between the languages, typing is most important."

One conclusion of the study is that weakly typed languages are superior to strongly typed languages for the design of system structures that naturally reflect the domain of application: "A program designed in a weakly typed language can have a class structure that closely resembles the programmer's intuition about the program's structure ... A program designed in a strongly typed language has its classes defined so that the type system can check the relations between them—but this design can distort the original, intuitive class structure."

However, this structural flexibility comes at the price of brittleness. This brittleness significantly hampers the ultimate usability of the system, as noted by Wolf: "Many bugs in weakly typed programs can be found only by executing the programs on the proper input set. The result is that weakly typed programs tend to have more chronic bugs than strongly typed programs. .. Weakly typed programs tend to start working much more quickly, but because bugs continue to be found long after the program is in operation, the programs are not as useful as they could be." Corroborating evidence for the brittleness of weakly typed systems comes from a study in [Gan77].

If weakly typed languages are good for design, and strongly typed languages

are good for robust implementations, then a natural solution is to combine them. However, Wolf's study concludes that this solution raises problems of its own: "Why not create a prototype program in a weakly typed language and then create a more robust production version in a strongly typed language? ... [Because] programs must be redesigned for strongly typed languages. The redesign cost raises a barrier inhibiting programmers from re-implementing their prototypes in safer, strongly typed languages."

This study brings into sharp relief the dichotomy between strongly and weakly typed programs, and the hard choice that must be made at the language-level: "My experience with Flavors and C++ shows that the choice of a language can have a profound influence on program design in several ways. Programmers are wise to consider these issues carefully when choosing a language for a programming project."

Type flow analysis for exploratory software development provides a means to go between the horns of this dilemma. Rather than being forced to choose between flexible-but-brittle or constrained-but-robust, type flow analysis allows programmers to incrementally move the implementation anywhere along a continuum between these two extremes. If programmers eventually desire robustness equivalent to that of a strongly typed system, they can structure the implementation into a form for which type-level correctness can be automatically assessed. While this will incur some developmental overhead, it will certainly be less than that foreseen in Wolf's study, since reimplementation in a different language is not required. If structural flexibility outside the capabilities of the analysis mechanism is required, this type level information may be asserted rather than inferred (or the anomolies reported may be ignored) and a correspondingly less rigorous form of assessment will result.

Research such as the study by Wolf, as well as the other research on exploratory software development referred to in Chapter 2, indicates the promise of the general goals of type flow analysis. However, the question remains: how well does type flow analysis work in its current incarnation? The next section provides some answers to this question.

5.2 Evaluating ESSIE, an implementation of type flow analysis

As described in Chapter 3, ESSIE is an implementation of type flow analysis that provides a language for defining control flow and data flow at the type level, using a type expression language similar in concept to that employed in basic parametric polymorphic type inference, but with augmented facilities for expressing the control and data flow of these objects through the system under analysis. ESSIE currently consists of about 10K lines of Lisp code, defining 80 analyzers and 16 analysis objects.

ESSIE has been exercised against various Lisp functions and small scale systems to test its ability to accurately model the type-level structure of exploratory software, to uncover errors, and to provide a new form of type-level documentation. This section presents findings from the use of ESSIE on a representative Lisp system called PTI, a Lisp-based implementation of the basic polymorphic type inference algorithm described in [Car87]. PTI consists of about 50 Lisp functions and approximately 500 lines of code, and was implemented by the author before the current research on type flow analysis began. It contains various forms of parametric as well as ad-hoc polymor-

phism, side-effects, and typical Lisp programming idioms. The source code for PTI along with ESSIE's analysis is presented in Appendix A. Declarations used by ESSIE are shown in Appendix B. Appendix C shows the execution strand-level structural representations obtained by ESSIE for this system.

5.2.1 Overview of the findings

The use of ESSIE on PTI demonstrated the ability of type flow analysis to support the process of exploratory development. This support includes the following:

- ESSIE was able to locate a previously unknown bug in the PTI system. The `extend-nongenvars` function is supposed to accept either a type variable or a list of type variables, and return a new list of type variables. ESSIE documented that the function could not accept a type variable as an argument, revealing the erroneous use of `symbolp` in the function where `t-var-p` was required.

- ESSIE documented previously unknown type-level characteristics of functions. In the case of `occurs-in-p` discussed below, it indicated a degree of polymorphism in the function that was an unintentional artifact of the particular implementation.

- ESSIE appeared to type the function `analyze-application` incorrectly. However, upon closer inspection of the function, it was determined that the analysis provided by ESSIE was not in error, but rather the structure of the code was subtle and prone to incorrect use. The restructuring re-

quired for a more appropriate type flow analysis improved the robustness of the code.

In addition to these positive findings, the analysis of PTI also uncovered interesting limitations of the mechanism, providing a source of future research directions.

For example, ESSIE is clearly a prototype implementation that is useful primarily as an evaluation mechanism for the concepts of type flow analysis. In its current form it is not, however, generally useful as a Lisp type assessment tool.

Some of the reasons for this are mundane—such as the lack of a sufficiently high-level user interface. For example, the user should be able to invoke a single command to perform the analysis of an entire system. ESSIE provides only an expression-level analysis command, which leaves it to the user to order a sequence of invocations of this command on the system's functions so as to traverse the system's calling hierarchy.

While such user interface additions are straightforward to provide, there are other, deeper reasons why ESSIE falls short of an adequate type assessment tool. These reasons fall into two major categories:

- ESSIE's current type expression language is limited in its expressiveness, particularly for heterogeneous structures.
- ESSIE's control and data flow mechanisms focus almost solely on the type level, while some additional value level representations are occasionally required for precise type-level structural assessment.

The principal limitation of the current type expression language is its limited support for heterogeneous data structures. The heterogeneity of global data structures can be approximated in ESSIE by declaring a set of strands as the result type that enumerate the element types in the structure. For example, to declare that a global variable `player-numbers` is a list of integers and strings, one would invoke the following:

```
(define-symbol-analysis player-numbers
  :constraints (and (listp foo)
                    (integerp (car foo))
                    (stringp (car foo))))
```

This effectively defines the variable `player-numbers` to be a function of no arguments that returns two strands, one with a list[string] result type and one with a list[integer] result type. Each time `player-numbers` is referenced the surrounding context must accomodate two possibilities: that its type-level value is either a list of strings or a list of integers. This approximates, but does not precisely model, an actual list of integers or strings, since ordering information within the list is lost.

While heterogeneous structures can be declared, the current implementation of type flow analysis does not provide any mechanism to infer such types, which means that the type-level value of heterogeneous uses of functions like `list` must be declared as well.

The principal problem with inferring heterogeneous data objects concerns unification: it is no longer possible to obtain a single set of substitution pairs representing the unification (or lack thereof) between two objects. Unification of heterogeneous objects becomes a "fuzzy" operation: there may be multiple ways to unify the objects, some of which may succeed, and others of which may

fail. Some of the preliminary design of this extension to ESSIE is described in the next chapter. Examples of this problem will be discussed in the description of the PTI system analysis presented below.

A second class of limitations of ESSIE concerns the granularity of control and data flow. One of the assumptions underlying the original design of type flow analysis is that representing control and data flow at the type-level *is* sufficient to provide precise characterizations of functional and data types, and sufficient to detect structural inconsistencies with a relatively low "signal-to-noise" ratio.[1] This assumption is common to all type checking mechanisms, be they inference or declaration based. A surprising result of this evaluation is that for a language like Lisp, such an assumption is often not warranted, and representation of control and data flow at the value level may occasionally be required. This need arises from Lisp's support for *assignment polymorphism*: the values of variables, as well as functions, are allowed to be polymorphic. Manifestations of this issue will occur and be discussed in the context of PTI below.

The evaluation of ESSIE demonstrates that additional problems must be resolved before the goal of a true software development tool is realized. However, it also demonstrates that the fundamental direction is sound, that type flow analysis can support both the assessment of structural consistency and evolution away from structurally ambiguous systems, and that the representations and analysis techniques currently implemented provide a solid foundation for future improvements. The next subsections highlight examples from the

[1] Although some value-level representations, such as true, false, and `nil` are present in ESSIE.

• *The defstruct declaration for type variables:*

```
(define-defstruct-analysis t-var
    :conc-name "T-VAR."
    :constructor (make-t-var (instance))
    :constraints (or (null instance)
                     (t-var-p instance)
                     (t-op-p instance)))
```

Figure 31: The global declaration of T-VAR

analysis of the PTI system that provide particularly interesting insights into the strengths and weaknesses of type flow analysis.

5.2.2 Assignment polymorphism and structural misinterpretation

A classic polymorphic variable in Lisp is `nil`, which could return a value that could be interpreted as either a symbol or a list. The function `null` is a predicate that returns true if it is passed `nil`, and false otherwise. The analysis of PTI revealed how the assignment polymorphism of `nil` can combine with `null` to lead to an incorrect representation of PTI's structure by ESSIE.

Remember that the first step in analyzing a system using ESSIE is to declare the type of global structures. In PTI, one of these global structures is called `t-var`. The instance slot of the `t-var` structure can hold one of three types of objects: `nil`, another `t-var` structure, or a `t-op` structure. The declaration of this global structure is shown in Figure 31.

While the declaration of `t-var` appears reasonable, it actually leads ESSIE to the conclusion that the instance slot may be filled by any arbitrary list, in

addition to type variables or type operators. This is due to the interpretation of `null` as succeeding not only for `nil` interpreted as a symbol, but also for `nil` interpreted as a list.

The solution adopted in ESSIE is to require the programmer to disambiguate these two senses by using different predicates: the Lisp predicate `endp` supports the operation on list structures, while the interpretation of `null` is restricted to testing for the presence of the `nil` value. The impact of this modification on the analysis of the PTI system was minimal: only one change was required in the function `occurs-in-p`, where `null` rather than `endp` was used to test for the end of a list.

Requiring the programmer to use the "right" predicate when more than one is possible appears a reasonable solution to these structural ambiguities. Even though the analysis system interprets `null` in a more restricted sense than normal, the original interpretation can always be recovered through testing for either of the disambiguations. For example, to recover the original interpretation of `null` at the analysis level, the programmer simply uses `(or (null x) (endp x))`.

Assignment polymorphism is strictly prohibited in functional polymorphic languages, as well as in imperative strongly typed languages. The analysis of PTI helps reveal the cost of including such a capability in the language in terms of greater structural ambiguity and possibility of misinterpretation. On the other hand, assignment polymorphism, like any other polymorphism, supports genericity and abstraction. While this research identifies some of the problems associated with assignment polymorphism, future research is needed to assess what corrective measures, if any, should be taken.

While requiring developers to be more careful in their use of functions like `null` reduces analysis problems, it doesn't eliminate them. `Null` also appears in contexts in PTI that demonstrate more fundamental limitations of representing structure at the type level.

5.2.3 Type-level and value-level structure may be closely entwined

Like all other analysis mechanisms at the type level, type flow analysis does not attempt to preserve value-level information about data and control flow. For both imperative and functional strongly typed languages, value-level information is irrelevant for assessing the type-level structural consistency of the program. An interesting outcome of the analysis of the PTI system is that, in some cases at least, value-level information is necessary to understand the type-level structure.

Consider the function **prune** (Figure 32), which is part of a unification mechanism in PTI. Recall that part of the inference process requires unifying two representations of an identifier gathered from different contexts, and that this unification may result in the instantiation of type variables from one or both of the representations. In PTI, `t-var` is the structure that represents polymorphic type variables, and `t-op` (for "type operator") is the representation used for everything else—integers, strings, lists, etc. Instantiation of a type variable is accomplished by setting its instance field to a new type expression, which could be either another `t-var` (which could itself be instantiated), or a `t-op` (which cannot be itself instantiated, though it may have internal structure that could be instantiated.) The process of unification thus builds up "chains" of instantiations of type variables to other type variables, possibly

• *The prune function for traversing chains of instantiated type variables:*

```
(defun prune (type-exp)
  "Returns a 'pruned' version of TYPE-EXP"
  (cond
    ((t-op-p type-exp)
     type-exp)

    ((and (t-var-p type-exp) (null (t-var.instance type-exp)))
     type-exp)

    ((t-var-p type-exp)
     (prune (t-var.instance type-exp)))))
```

Figure 32: The definition of PRUNE

terminated by an instantiation to a type operator. The function of prune is to return the end of such a chain when passed an arbitrary type expression (i.e. a t-var structure or a t-op structure).

ESSIE signals a type error when processing this function. It results from the fact that one of the potential recursive invocations of prune fails to satisfy any of the conditional clauses. The root of this error is exposed by examination of the execution strands generated just after processing the function, but before the resolution of the recursive strands into the base strands. These strands are shown in Figure 33. Recall that the processing of recursive functions has two parts. In the first part, the function is processed as if there were no recursion: recursive invocations simply store the argument types passed, and a special contextual type variable is returned as the result of the recursive invocation. In Figure 33, each contextual type variable is represented by the symbol Ctv.

• *The execution strands for prune:*

```
FC: ((T-OP-P TV-1356))
CE: < (PRUNE {RFn})(TYPE-EXP TV-1356) >
RT: TV-1356

FC: ((T-VAR-P T-Var-1360) (NOT (T-OP-P T-Var-1360)))
CE: < (PRUNE {RFn})(TYPE-EXP T-Var-1360) >
RT: T-Var-1360

FC: ((T-VAR-P T-Var-1366) (NOT (T-OP-P T-Var-1366)))
CE: < (PRUNE {RFn <AR: (Nil) RT: Ctv> })(TYPE-EXP T-Var-1366) >
RT: Ctv

FC: ((T-VAR-P T-Var-1372) (NOT (T-OP-P T-Var-1372)))
CE: <(PRUNE {RFn <AR: (T-Var-1369) RT: Ctv>})(TYPE-EXP T-Var-1372)>
RT: Ctv

FC: ((T-VAR-P T-Var-1378) (NOT (T-OP-P T-Var-1378)))
CE: <(PRUNE {RFn <AR: (T-Op-1375) RT: Ctv>})(TYPE-EXP T-Var-1378)>
RT: Ctv
```

Figure 33: The strand set for PRUNE

Once the entire function is processed and the set of execution strands illustrated in Figure 33 has been generated, the second part, called recursive resolution, begins. In this part, the set of execution strands is examined for strands that contain recursive function invocations, and these recursive references (the inductive cases) are resolved against the set of strands without recursive references (the base cases). The resolution process consists of ensuring that each recursive strand can be extended along at least one non-recursive strand.

In Figure 33, the first two strands are the non-recursive cases (note that

no argument/return value information has been collected in the constraint environment), whereas the last three strands indicate three different potential recursive invocations of prune—one with a `nil` argument, one with a `t-var` structure argument, and one with a `t-op` structure argument. Therein lies the problem—in actuality, prune can only be recursively re-invoked with either of the two structures as arguments: the `nil` case is filtered out by the second conditional clause in prune.

ESSIE produces three recursive re-invocations because the instance slot of `t-var` is declared to hold any of three types of values, as we saw before in Figure 31. Thus, any time the `t-var.instance` slot accessor function is invoked, ESSIE returns three possible return values. It is for this reason that ESSIE attempts to resolve a recursive invocation of prune with the `nil` value against the base strands, which only accept `t-var` and `t-op` structures, resulting in an error.

While this analysis error also has its roots in ESSIE's inability to interpret the assignment polymorphism of `null` correctly, it cannot be solved by disambiguation: the problem is that `null` is an operation at the value-level, not the type level. If `null` was doing a type-level computation, then it would be a type predicate, and this issue could be ameliorated using the current analysis mechanisms.

However, what prune reveals is that completely removing this form of imprecision from type flow analysis requires modelling not only the type structure of functions, but also the types of values stored into memory locations during execution. In the limit, such an analysis essentially requires ESSIE to do a full-fledged value-level symbolic execution of the program, with all the accompaning problems that value-level processing incurs.

The current implementation of ESSIE handles this class of error simply by printing a warning message indicating its occurrence. If going to full fledged value-level symbolic execution system is the strictest remedy for this error, then the current behavior is certainly the most lenient. Several intermediate solutions are also possible: the programmer could add a declaration to the recursive invocation of prune asserting the "reduced polymorphism" of the slot accessor function for that particular point in the source code. Alternatively, it might be possible to eliminate a large percentage of these errors through restricted classes of value-level symbolic execution, expanding upon the current processing of booleans. Exploration of this issue is an ongoing subject of research.

5.2.4 Redundant, but not identical, execution strands

A more minor problem is revealed by the final analysis of prune, illustrated in Figure 34. As its function implies, the type level structure of prune is simple: given a type variable, it could return either a type variable or a type operator, and given a type operator, it must return a type operator. Although this representation only requires three strands, four are generated by ESSIE, with the second differing from the fourth only due to the sharing of structure between the parameter and result analysis objects.

To understand how this occurs, one must compare the unresolved set of strands for prune in Figure 33 against the resolved set of strands in Figure 34. The second strand from the unresolved set, which is a base case of the recursion, is also the second strand of the resolved set. The fourth strand of

- *The abbreviated representation for* prune*:*

```
Function PRUNE
 < (TYPE-EXP T-Op-15) > ==>> T-Op-15
 < (TYPE-EXP T-Var-5) > ==>> T-Var-5
 < (TYPE-EXP T-Var-8) > ==>> T-Op-9
 < (TYPE-EXP T-Var-12) > ==>> T-Var-13
```

- *The full strand-level representation for* prune*:*

```
  FC: ((T-OP-P TV-15))
  CE: < (TYPE-EXP TV-15) >
  RT: TV-15

  FC: ((T-VAR-P T-Var-5) (NOT (T-OP-P T-Var-5)))
  CE: < (TYPE-EXP T-Var-5) >
  RT: T-Var-5

  FC: ((T-VAR-P T-Var-8) (NOT (T-OP-P T-Var-8)))
  CE: < (TYPE-EXP T-Var-8) >
  RT: T-Op-9

  FC: ((T-VAR-P T-Var-12) (NOT (T-OP-P T-Var-12)))
  CE: < (TYPE-EXP T-Var-12) >
  RT: T-Var-15
```

Figure 34: The final analysis of PRUNE

the resolved set (the redundant strand) is generated from the resolution of the fourth unresolved strand against the second unresolved strand.

While ESSIE removes duplicated strands whenever possible, it cannot in this case, since the two strands, though operationally equal, are structurally different. Although this situation occurs relatively infrequently, it could still result in a significant source of analysis-time overhead (note that this single extra strand adds 33% more structural overhead to any function calling prune).

There are several potential courses of action. Since the redundant strand only results in a performance penalty, the user is free to simply leave it alone. On the other hand, since the strand is redundant, the user could also remove it without fear of weakening the type assessment machinery. In other words, removal of this strand does not correspond to a "declaration", since it does not assert new information. A third course of action is to implement a more sophisticated redundant strand detection mechanism, which removes not only structurally *isomorphic* execution strands, but structurally *homomorphic* execution strands as well—those whose differences do not make an impact on type flow analysis.

5.2.5 Recursive control flow lacking type invariance

As noted in Chapter 2, one of the major problems confronted by traditional symbolic execution systems is the need to infer a closed form expression for the results of iteration. In parametric polymorphic type inference, this problem is easily finessed: loops are simply required to be invariant at the type level. In other words, the type returned from a loop computation is invariant across the number of iterations through the loop.

In type flow analysis, a slightly modified version of the parametric polymorphism restriction was adopted: while the analysis of a loop does not attempt to characterize the type as a function of the number of times through the loop, as in the case of symbolic execution systems, it doesn't restrict the result of a loop computation to a *single* type as in parametric polymorphic type inference. Thus, a loop computation can return a set of types (or more accurately, a set of execution strands) as its result, but this *set* of execution strands should satisfy the loop invariance criterion. `Prune` is an example of a function whose (recursive) loop returns a set of strands (with result type of either `t-op` or `t-var`), though this return set is independent of the number of recursive invocations.

This "type set invariance" design decision seemed like a reasonable restriction that reflected a conventional wisdom in exploratory software development: loop-dependent types are normally the result of implementing complex data structures as lists, which should normally be eschewed for structure-based implementations.

A surprising result illustrated by the PTI analysis is that functions that are intended to operate in a type set invariant fashion may allow, in addition to the invariant execution path set, an additional set of execution paths whose result type *is* a function of the number of iterations. `Occurs-in-p` (Figure 35) is an example of such a function.

`Occurs-in-p` is a predicate that returns `t` or `nil` depending upon whether its first argument, a type variable, occurs in its second argument, which may be either a type variable, or a list of either type variables or type operators.

Figure 36 shows the analysis results for `occurs-in-p`. The first eight strands represent the expected parameter/result combinations, with the first

- *The* occurs-in-p *function checks to see if t-var occurs in t-exps:*

```
(defun occurs-in-p (t-var t-exps)
  "Check to see if T-VAR occurs in T-EXPS."
  (let ((t-var (prune t-var))
        (t-exps (if (listp t-exps)
                    t-exps
                    (prune t-exps))))

    (when (t-var-p t-var)
      (cond ((t-var-p t-exps)
             (eq t-var t-exps))

            ((and (listp t-exps) (endp t-exps))
             'nil)

            ((t-op-p t-exps)
             (occurs-in-p t-var (t-op.args t-exps)))

            ((listp t-exps)
             (if (occurs-in-p t-var (first t-exps))
                 'T
                 (occurs-in-p t-var (rest t-exps))))))))
```

Figure 35: The definition of OCCURS-IN-P

• *This analysis indicates that T-EXPS can be passed an arbitrarily nested list of type expressions:*

```
Function OCCURS-IN-P
(T-VAR T-Var-141)(T-EXPS T-Op-142) ==>> Boolean[T]
(T-VAR T-Var-150)(T-EXPS T-Op-151) ==>> Boolean[F]
(T-VAR T-Var-159)(T-EXPS T-Var-160) ==>> Boolean[T]
(T-VAR T-Var-168)(T-EXPS T-Var-169) ==>> Boolean[F]
(T-VAR T-Var-117)(T-EXPS LIST[ T-Var-118]) ==>> Boolean[F]
(T-VAR T-Var-131)(T-EXPS LIST[ T-Var-132]) ==>> Boolean[T]
(T-VAR T-Var-7)(T-EXPS LIST[ T-Op-8]) ==>> Boolean[T]
(T-VAR T-Var-17)(T-EXPS LIST[ T-Op-18]) ==>> Boolean[F]
(T-VAR T-Var-88)(T-EXPS LIST[ LIST[ TV-89]]) ==>> Boolean[T]
(T-VAR T-Var-102)(T-EXPS LIST[ LIST[ TV-103]]) ==>> Boolean[F]
```

Figure 36: The analysis of OCCURS-IN-P

parameter always a type variable, the second parameter taking on the various combinations of type expressions and lists of type expressions, and the result alternating between the two boolean values.

However, in addition to these first eight expected execution strands, the analysis of `occurs-in-p` also includes two additional, unexpected strands. These strands indicate (but do not precisely represent) the fact that `occurs-in-p` can also accept a second argument that is a list of a list of type expressions. This representation is approximate because of the way recursion is processed in ESSIE: the resolution process models the effect of going through the loop a single time, which is sufficient under the assumption of invariance. In this case, one time through the loop is sufficient to establish that a list of lists can be accepted, but cannot place any constraints on the internal structure of the embedded list.

Upon re-examination of occurs-in-p in light of this analysis, it becomes clear that the function is "accidentally" able to process arbitrarily deeply nested lists of type expressions, and the function will simply recursively traverse however many levels of nesting exist until it comes to a list of type expressions to process.

Several conclusions can be drawn from the analysis of `occurs-in-p`. First, it demonstrates the ESSIE is capable of revealing rather subtle characteristics of polymorphic functions, and thus can serve as a useful design and documentation aid. Second, although ESSIE is incapable of *accurately* typing functions whose type structure is a function of the number of iterations, it is capable of *indicating* the presence of such type structure, and providing developers with useful information from which the precise type structure can be diagnosed.

On the downside, however, the actual analysis provided by ESSIE is simply incorrect: `occurs-in-p` cannot execute on an invocation where TYPE-EXPS is bound to a list of a list of integers, even though the execution strand set indicates this to be a valid argument. There are several approaches to rectifying this difficulty.

The first is to force ESSIE to signal an error when such functions are encountered, which is signalled through the success of the unification occurs check. ESSIE, although it allows developers to supply an occurs-checking function in the definition of analysis objects, does not require it, and defaults the occurs check to a constant function that returns `nil`. The set of analysis objects used to process PTI all employ this default, which is why ESSIE did not raise an error when processing this function. Under this approach, the developer would explicitly need to test (for example, as part of the `when` condition) that the

TYPE-EXPS argument was a list containing only type variables and type operators. The advantage of providing the occurs check is safety: it removes the possibility that ESSIE could provide an incorrect type for a function; however, it also restricts the kinds of functions for which ESSIE will return any form of type representation.

A second approach is to let the analysis of such functions succeed, but manually prune the set of offending execution strands from the resulting analysis. Since pruning strands does not constitute a declaration of the function's type, the ensuing analysis based upon this modified information is not subject to an assertion made by the developer. On the other hand, this approach requires the developer to somehow catch all instances of such functions.

Perhaps the best approach is to combine the above: add the occurs checking functions to the analysis object definitions in ESSIE, but modify their use so as to signal a warning, rather than an error, when the occurs check fails. The warning would indicate to the developer which functions' analyses are suspect, and allow the developer to then inspect the analyses and either modify the code or modify the analysis as desired.[2]

5.2.6 Functional arguments and functional analyzers

ESSIE provides two mechanisms to define the structure of functions. The first representation is a braid, or set of execution strands, generated either through the analysis of the function's definition, or through the declaration of the function's type with `define-fn-analysis`. The second representation is

[2]It is ironic, or perhaps simply appropriate, that this issue with occurs checking in ESSIE occurred during the analysis of the occurs checking function for PTI!

through the use of `define-analyzer`, which allows the user to "escape" out of the conventional type flow application mechanism, in much the same way that Lisp macros allow the user to escape out of the normal Lisp function argument evaluation mechanism. `Define-analyzer` is necessary not only to define the type flow processing of control flow constructs such as `let` or special forms like `defun`, but also to bootstrap the system with a core set of predefined Lisp functions, such as `car`, `cdr`, `list`, and so forth.

One problem that arose during the processing of PTI occurred in functions like `retrieve-type`, which contained the following call to `find`:

```
(find identifier env :test #'car)
```

Here ESSIE signalled an "undefined function" error: `car` was not known to the system. Of course, the system knows all about `car`— it's just that that information is encoded procedurally, as an analyzer, rather than declaratively, as a braid. While ESSIE supports the passing of functional objects as parameters, it only "understands" the representation of functional objects in their declarative form.

Since the ability to pass around primitive Lisp functions is obviously desirable for an analysis system, some sort of solution was required. The most general solution is to extend the type flow application mechanism to support both procedural and declarative representations of functions. The solution adopted for PTI is far more perspicuous, though perhaps not as general. Figure 37 illustrates the general strategy.

This "trick" results in the system automatically translating the procedural, analyzer-based representation of `car` into its declarative, braid-based represen-

```
(define-fn-analysis car
  #'(lambda (x) (car x)))
```

Figure 37: Obtaining a declarative from a procedural function representation.

tation. The interior call to car within the lambda form is processed by the analyzer, but the analysis returned for the lambda form is a braid, which is then stored as the declarative representation of car, and retrievable by the type flow analysis mechanism.

Although this technique works well for producing a declarative representation for the "boot-strapping" functions that have normal Lisp evaluation semantics, it does not suffice for Lisp forms like let or defun, where a truly procedural interpretation is necessary. On the other hand, such special forms are not allowed as functional arguments, so it is not clear that a declarative mechanism is required. Future research and experience with the system is required to determine the extent to which this translation technique solves the issues that arise in maintaining both procedural and declarative representations for functions.

5.2.7 Structural ambiguity as structural critique

Assignment polymorphism can lead ESSIE to ambiguous interpretations some of the time, and incorrect interpretations of the code at other times. An interesting finding illustrated by the analysis of PTI is that structural ambiguity or misinterpretation may also reflect code that could be written

• *An analyzer function in PTI:*

```
(defun analyze-application (appl env nongenvars)
  "Return a type expression for a function application."

  (let* ((name (car appl))
         (arg-list (cdr appl))
         (function-type (analyze-exp name env nongenvars))
         (arg-list-types (mapcar #'(lambda (arg)
                                     (analyze-exp arg env nongenvars))
                                 arg-list))
         (result-type (make-t-var nil)))
    (unify-type function-type
        (make-t-op
          'function
          (append arg-list-types (list result-type))))

    result-type))
```

In the revised version of this function after type flow analysis, the final reference to `result-type` *is replaced by a call to* `(prune result-type)`.

Figure 38: The definition of ANALYZE-APPLICATION

in a clearer and more robust manner. The function `analyze-application`, shown in Figure 38, illustrates this point. The purpose of this function is to return a type expression representing the result of analysis of an expression.

The type flow analysis of this function is surprising because the result type is simply a type variable, rather either a type variable or a type operator. This result seemed to indicate a bug in ESSIE, until closer inspection of the structure of the `analyze-application` function revealed the reason for this result.

Indeed, `analyze-application` does return a type variable—at least—a

structure of that type is returned. However, the *interpretation* of this object is different depending upon whether or not its instance field is instantiated to another type variable or type operator. The "semantic" object returned from `analyze-application` is whatever the *pruned* value of the type variable bound to `result-type` is, which could be either a type variable or a type operator.

Obviously, such value-level interpretation is far beyond the inference capabilities of the type flow analysis mechanism. The analysis results did, however, raise a software engineering issue: the code in its original form, while not necessarily in error, still leaves it to the context invoking `analyze-application` to invoke `prune` in order to properly interpret the result type. Such a design decision seems clearly error-prone, and the clear solution is to return not `result-type`, but the results of invoking `prune` on result type. After replacing the final occurrence of `result-type` with `(prune result-type)` in the function, its analysis results in the anticipated set of strands providing both type operator and type variable results, as shown in Figure 39.

5.2.8 Type flow analysis requires optimized unification algorithms

ESSIE uses unification mechanisms similar to those described in [Car87], which introduced them as "reasonably efficient and quite usable in practice for typechecking large programs." This is probably accurate for conventional parametric polymorphic type inference, but the use of ESSIE has shown that conventional implementations of unification may be inadequate for the type flow analysis of large scale software systems.

In parametric polymorphic type inference, the type level representation of

```
Function ANALYZE-APPLICATION:

<(APPL LIST[LIST[TV-40]])(ENV LIST[Env])(NONGENVARS LIST[T-Var]) >
   ==>> T-Op
<(APPL LIST[LIST[TV-44]])(ENV LIST[Env])(NONGENVARS LIST[T-Var]) >
   ==>> T-Var
<(APPL LIST[Sym])(ENV LIST[Env])(NONGENVARS LIST[T-Var]) >
   ==>> T-Op
<(APPL LIST[Sym])(ENV LIST[Env])(NONGENVARS LIST[T-Var]) >
   ==>> T-Var
```

Figure 39: The analysis of the revised ANALYZE-APPLICATION

an expression is simply a type expression (i.e., either a type variable or a type operator structure), and the process of application involves one unification per argument. However, in type flow analysis, the type level representation of an expression is a braid. Braids consist of a set of strands. Each strand contains a path condition (where each type predicate is matched with a type expression), a constraint environment (containing all the locally declared variables, each bound to a type expression), and a result type (a single type expression). The final braid for `occurs-in-p`, for example, contains 12 strands and 96 type expressions.

Consider the process of a single application of `occurs-in-p` to a set of arguments. First, a fresh version of its braid must be generated from the one stored in global memory, since the analysis process will destructively modify its type expressions. `Occurs-in-p` accepts 2 arguments. Assuming these arguments evaluate to braids with, say, 4 strands each, that makes 16 potential composite argument pairs that must be applied to the 12 strands in

`occurs-in-p`. From these 16 potential composite pairs, assume that half of them are actually feasible and structurally distinct from one another. That means that 7 more copies of the braid for `occurs-in-p` will be required—one for each potential execution path through the function. In other words, almost 800 type expressions will be generated just to accomodate the role of `occurs-in-p` in a single application context.

As a result, it is fortunate that this research was attempted in the era of 100 megabyte virtual memory spaces, since the creation of 20 to 30 thousand type expressions is typical for the kinds of functions encountered in the PTI system. Largely as a result of this creation overhead, the analysis time for a function like `occurs-in-p` is commonly around 20 seconds on a Texas Instruments microExplorer Lisp Machine.

While these numbers seem to present real problems for scaling up the prototype to the analysis of real systems, the situation is not quite as bad as it appears. Unification simply generates a great deal of garbage—i.e. reclaimable memory—and the current implementation does not exploit that feature of the algorithm. While 20-30K type expressions might be generated in the process of analyzing a function like `occurs-in-p`, only 96 of them are actually needed once the job is done. According to a recent survey article [Kni89], one of the current trends in unification research is toward more efficient implementations of the algorithm. Future research on ESSIE may be able to exploit these advances to cut down significantly on the unification overhead incurred in type flow analysis.

5.3 A type-level study of GBB

The results of the type flow analysis of PTI indicate that the representations and mechanisms of ESSIE form a good basis for describing type-level structure, but these results also leave several questions unanswered. Chief among them is the question of whether type flow analysis is suitable for, and scalable to, real world exploratory systems. To answer that question, a study of the types of objects typically passed to and returned from such a real world exploratory software system, called GBB [CGM86], was undertaken.

GBB is a high-performance blackboard application generation system, which has successfully made the transition over the past 5 years from an experimental research vehicle to its current form as a robust, commercially marketed software system. GBB has been ported to over a half dozen hardware platforms and has been distributed to over 300 university and industry sites.

The best way to answer the question of whether type flow analysis is appropriate for real world exploratory software would be to simply analyze its structure using ESSIE. Unfortunately, the current facilities of ESSIE fall short of those required for the analysis of GBB, which uses a much broader range of Lisp features than are accomodated in the current version of the prototype. Even if those features were implemented, however, it would not be clear whether type flow analysis would be necessary and sufficient for providing useful characterizations of the type level structure.

For example, perhaps real-world exploratory software simply eschews Lisp's capabilities for ad hoc polymorphism, deeming them too error-prone for such development (much in the same manner that a labelled GOTO facility exists

in Common Lisp, but rarely appears in code other than that produced in introductory programming classes.) ESSIE might be capable of analyzing such software, but its distinguishing mechanisms, such as type predicates and path conditions would be superfluous. Fortunately, the study of GBB demonstrates that ad hoc polymorphism forms a significant proportion of its polymorphism.

While some evaluation of the necessity of mechanisms like type flow analysis for real world exploratory software is possible without a fully functional implementation, evaluating its sufficiency is much more problematic. However, it is possible to test for negative evidence of its sufficiency.

For example, if exploratory software manipulates a large number of highly heterogeneous and/or recursive list structures, then a more sophisticated representation for such structures than that currently implemented would appear mandated. In addition, if exploratory software was *too* ad hoc polymorphic, where each function's representation required dozens or hundreds of execution strands, then a combinatorial explosion is possible.

Part of the data gathered during the study of GBB provides some insight into both of these situations. While the answers obtained are not completely clear-cut, they appear to support the notion that type flow analysis can be scaled to real world exploratory software. For example, 83% of the functions studied appear to require only 3 or fewer execution strands for their representation, and only 7% of the lists were heterogeneous with more than two elements.

The next subsections describe the study of GBB and the results in more detail. Following this, the chapter concludes with a summary of the evaluation of type flow analysis and ESSIE.

5.3.1 Methodology

If an actual type flow analysis of GBB could be accomplished, the representation of functions would consist of the range of types that each function could accomodate and return, as well as the conditions under which a particular set of argument and return types would be feasible, based upon the definition of the function itself. The actual study of GBB performed a kind of "shadow" of type flow analysis, retrieving data on the range of types passed and returned from each function during *actual* executions of the GBB system. Since the functions' definitions were not analyzed, but only their run-time behavior, no path condition information was collected, and the type information collected for a function potentially represented a subset of the actual types accomodated. Nevertheless, this approximation to type flow analysis still revealed interesting structural characteristics of GBB.

The data collection employed the Lisp `advise` facility, which allows the "wrapping" of code around arbitrary Lisp functions. This code can access and alter both the values passed to the advised Lisp function, as well as the values returned from the execution of the function. Each function in the GBB system was advised with code that recorded, for each invocation of the function, the type of value passed to and returned from each parameter. This type information was itself an approximation in some cases. For example, lists were typed only with respect to their first level of structure, without respect to ordering, so that a list of alternating integers, strings, and lists of symbols would be typed simply as a list of integer, string, and list. In addition, `t` and `nil` were represented as individual, distinct types, even though the type of `t` is actually a symbol, and the type of `nil` is either a symbol or a list. This alternative was chosen to facilitate the identification of predicate functions.

• *This function builds a blackboard storage structure in GBB:*

```
(defun BUILD-BB-STORAGE (nodes)
  (cond ((space-instance-p nodes)
         (build-space-storage nodes))
        ((database-node-p nodes)
         (map nil #'build-bb-storage (db-node.vector nodes)))
        (t (map nil #'build-bb-storage nodes))))
```

• *The function produces the following analysis information:*

```
Function: BUILD-BB-STORAGE was invoked 3 different ways.
NODES : (LIST-OF DATABASE-NODE SYMBOL) ==> NIL
NODES : DATABASE-NODE                  ==> NIL
NODES : SPACE-INSTANCE                 ==> SPACE-INSTANCE
```

Figure 40: The definition and run-time behavior of BUILD-BB-STORAGE

Figure 40 illustrates an example GBB function and the information gathered from its analysis.

Run-time information was collected on 357 functions in the "core" of GBB[3] during the execution of two relatively simple blackboard applications. The first consisted of an example file containing approximately 70 invocations of GBB facilities, which exercises and documents the principal features of the system. The second application, called `simple-shell`, implements a basic agenda-based blackboard system. These two applications together resulted in 837 distinct type-level argument and parameter invocations of the 357 functions.

[3]A running system actually contains many more GBB functions, since the definition of blackboard structures results in GBB defining many additional functions specialized to operate on these objects. The analysis focused only on those functions that are independent of any particular GBB application.

Table 1: GBB functions, classified by number of distinct invocations.

Invocations	Number of Functions	*Invocations*	Number of Functions
1	169	*8*	7
2	92	*9*	6
3	37	*10*	4
4	25	*11*	0
5	4	*12*	0
6	8	*13*	1
7	4		

The conclusions drawn from this run-time data collection are summarized below.

5.3.2 Ad hoc polymorphism in GBB

GBB clearly contains both parametric and ad hoc polymorphism. An easy way to see that some form of polymorphism is occurring is by simply classifying the functions according to the number of their distinct type-level invocations, as shown in Table 1.

This table shows that 169 GBB functions were invoked with only one combination of argument and result types, 92 with two combinations, and so on up to the function `get-unit-description`, which was invoked with 13 different combinations of argument and result types. Out of 357 functions, therefore, at least 188, or roughly half are either parametric or ad hoc polymorphic.

Breaking down these 188 functions further requires deciding on an appropriate interpretation of parametric polymorphism for Lisp. For example, one reasonable interpetation is that a function is parametrically polymorphic if and only if its type can be expressed with the parametric polymorphic type

expression language. Under this interpretation, only 10% of the 188 functions are parametrically polymorphic, since heterogeneous lists cannot be expressed in that language.

One could also break down the functions by defining criteria for what guarantees ad-hoc polymorphism from the gathered data. Again, heterogeneous lists cloud the issue somewhat, but a correspondingly conservative definition to the one employed for parametric polymorphism would be: those functions whose arguments are passed two different types of objects, where lists of any sort are counted as a single type. In other words, functions whose invocations differ from each other only in terms of internal list structure would not be considered ad-hoc. Under this interpretation, 77 of 188 functions are ad-hoc, or around 40%.

These two interpretations indicate that ad-hoc polymorphism accounts for anywhere from 40% to 90% of the polymorphism present in GBB. Whatever the actual number is, it is clear that ad-hoc polymorphism plays a significant role in the polymorphism of GBB as a whole, and that for this instance of a real world exploratory software system, representational mechanisms akin to type flow analysis are necessary for the type-level structural description of the system.

Table 1 also lends support to the notion that the number of execution strands per function in GBB would be reasonably small. As the number of invocations increases in the table, the functions become predominantly parametrically polymorphic. `Get-unit-description`, the function with the most distinct type-level invocations of all, is a good case in point. As shown in Figure 41, this function is invoked during the study in 13 different ways. However,

Table 2: List occurrences, classified by internal element types.

Number Internal Element Types	*Number Occurrences*	*Percentage*
0	280	26%
1	237	22%
2	484	45%
3	74	7%

inspection of the actual function definition shows that only three execution strands would be required to represent this function's type level structure.

5.3.3 Heterogeneity in GBB

To get some sense for the use of heterogeneous structures in real world exploratory software development, every occurrence of a list either passed as an argument to or returned from a function during the analysis was collected. These 1075 uses of list structure were then classified according to their "degree of heterogeneity"—the number of distinct types of objects in their first level of structure. These results are presented in Table 2.

Lists with 0 internal element types are the empty list (`nil`), lists with 1 element type are homogeneous lists[4] and lists with 2 or more internal element types are heterogeneous. The results here are less conclusive than those presented for ad hoc polymorphism, but still encouraging: the degree of heterogeneity in GBB is limited to lists of only 2 or 3 element types (at the top level of structure.) However, there are a lot of them: over half of the lists observed were heterogeneous.

[4]There was no occurrence of a list with a list as its only internal element, which could have been potentially heterogeneous, even though it had only one element at its top level.

• *An example of a parametric and ad hoc polymorphic function:*

```
(defun get-unit-description (unit &optional no-error-p)
  "Return the unit-description associated with UNIT."
  (cond
    ((typecase unit
       (basic-unit
        (gethash (unit-type-of unit) *unit-description-hash-table*))

       (unit-description
        unit)

       (symbol
        (gethash unit *unit-description-hash-table*))
       (no-error-p nil))
     (t (error "~s is not a unit, unit-instance, or unit-description."
               unit)))))
```

• *The analysis information collected for* `get-unit-description`*:*

```
Function: GET-UNIT-DESCRIPTION was invoked 13 different ways.
Arg/Results:
UNIT : HYP;    NO-ERROR-P : NIL ==> UNIT-DESCRIPTION
UNIT : KSI;    NO-ERROR-P : NIL ==> UNIT-DESCRIPTION
UNIT : QUEUE-UNIT; NO-ERROR-P : NIL ==> UNIT-DESCRIPTION
UNIT : KS;     NO-ERROR-P : NIL ==> UNIT-DESCRIPTION
UNIT : UNIT5; NO-ERROR-P : NIL ==> UNIT-DESCRIPTION
UNIT : UNIT4; NO-ERROR-P : NIL ==> UNIT-DESCRIPTION
UNIT : UNIT3; NO-ERROR-P : NIL ==> UNIT-DESCRIPTION
UNIT : UNIT2; NO-ERROR-P : NIL ==> UNIT-DESCRIPTION
UNIT : UNIT1; NO-ERROR-P : NIL ==> UNIT-DESCRIPTION
UNIT : SYMBOL; NO-ERROR-P : NIL ==> UNIT-DESCRIPTION
UNIT : SYMBOL; NO-ERROR-P : T   ==> UNIT-DESCRIPTION
UNIT : SYMBOL; NO-ERROR-P : T   ==> NIL
UNIT : UNIT-DESCRIPTION; NO-ERROR-P : NIL ==> UNIT-DESCRIPTION
```

Figure 41: The definition and invocation information for GET-UNIT-DESCRIPTION

While this data indicates that some accomodation for heterogeneous lists must be made in type flow analysis, more research will be needed to determine what sort of mechanisms will be required. A great deal depends upon the way in which heterogeneous lists are actually used: if they are primarily employed as simple, "temporary" data structures to hold a few objects during processing restricted to a single function, then a correspondingly simple, even declaration-based mechanism might be sufficient for the purposes of type flow analysis. On the other hand, if heterogeneous lists are long-lived objects that pass through much of the system and that display a complex, recursive, and strictly ordered structure, then better representations are mandated. Of course, which form heterogeneity takes may be based upon characteristics of the domain or of the developer's programming style.

Some of the heterogeneity in GBB is undeniably of the latter form, and arises from a useful feature of exploratory languages: the ability to represent source-level code as a data structure in the language. In the case of Lisp, of course, this data structure is a list. First, GBB implements a new set of language constructs for the definition of blackboard objects, which are represented (in the same manner as normal Lisp constructs) as complex, recursive, and ordered list structures. Second, GBB's role as an application generation system leads to the presence of many functions that incrementally "build" new function definitions by consing together pieces of list structure. Successful type flow analysis of systems with these features may necessitate better representations for heterogeneity.

GBB is but a single instance of an exploratory software system, and thus may not be entirely representative of the state of the practice. However, it is a good choice for this study, since it is an instance of an exploratory system that

fulfills the promise of the paradigm: having begun as a research prototype, it has evolved and matured into a robust, commercial grade product.

5.4 Conclusions

This chapter began by seeking corrobative evidence for the essential premise of the dissertation: that the use of type flow analysis can help programmers to obtain desirable qualities commonly associated with both weakly and strongly typed languages. The evidence presented consisted of the experiences of a researcher with the implementation of a system in both a weakly typed and a strongly typed language. Fortunately, the conclusions drawn from this study closely parallel the assumptions underlying this research: that weakly typed languages enjoy a structural flexibility that both allows software to more closely approximate the problem domain, as well as supporting ease of change and modification that leads to behaviorally evaluable systems with less developmental investment. The price paid for this is in the robustness of the resulting systems: strongly typed languages catch more errors early in the development process, and the absence of these errors is deemed significant enough that the researcher suggests that, in many cases, the obtained robustness offsets any structural disadvantages incurred. Based upon this evidence, type flow analysis aims in the right direction: toward providing a means to incorporate the same degree of type-level assessment that gives strongly typed languages their edge in robustness, while preserving the structural flexibility of weakly typed languages: in particular, their facilities for ad-hoc polymorphism.

The next section of this chapter addressed the question of how close ESSIE, the current implementation of type flow analysis, comes to the goals set out for type flow analysis. Clearly, ESSIE falls short of the goal of providing a robust and general type analysis tool for arbitrary styles of Lisp-based software development. What ESSIE accomplishes well is to illuminate the current status of the theory of type flow analysis, and to point out the directions in which further research must proceed. From that light, ESSIE appears to verify the essential promise of type flow analysis: even for a relatively simple system like PTI, ESSIE is able to:

- Point to a previously unknown type-level error—the use of `symbolp` in a function where `t-var-p` was required.
- Point to previously unrecognized type-level characteristics of a function—the ability of `occurs-in-p` to accept arbitrarily nested lists of type expressions.
- Point to a suspect function from a software engineering standpoint—the original version of `analyze-application`, which did not prune the type expression before returning it.

Just as important as these positive behaviors, though, are the problems encountered in using ESSIE, which delineate areas for future research. First, type flow analysis supports both parametric and ad hoc polymorphism in functions. However, another important form of polymorphism is assignment polymorphism, where the value stored in a memory location may take on one of many different types. ESSIE currently represents assignment polymorphism in a "referentially transparent" manner: the set of types retrieved from a

memory location is independent of the context in which the retrieval occurs. As illustrated by the function prune, such referential transparency is inadequate for modeling the real world, where only one type of value occupies a memory location at any one time. This result indicates that future research on ESSIE should include better representations for memory-based computation and for these ad-hoc values themselves.

The limited capabilities for representation of heterogenous objects also proved problematic during the analysis of PTI. While ESSIE provides a capability to approximate the type structure of globally defined heterogeneous types, this capability falls short of that required to completely handle the structures encountered in PTI. In this system, where heterogeneous lists are dynamically generated in a variety of contexts, ESSIE required cumbersome declarations that somewhat compromised the type assessment process. A topic of current research in type flow analysis is "fuzzy" unification, which would allow the type inference mechanism to represent such dynamically generated heterogeneous structures.

Finally, ESSIE exercises unification much more than parametric polymorphic inference mechanisms, and this means that algorithms and data structures deemed sufficiently efficient for parametric polymorphism may not be suitable for type flow analysis. Finding suitably optimized versions of unification for type flow analysis forms yet another subject for future study.

The final section of this chapter concerned real world exploratory software, and the question of whether type flow analysis is both necessary and sufficient for its analysis. An empirical study of GBB, a large scale exploratory software system, was undertaken to provide insight into this issue. The results clearly demonstrate that ad hoc polymorphism of the kind represented in this research

forms a significant structural component of GBB. This suggests that structural representations for this form of polymorphism are necessary and desirable. A second result concerns the heterogeneity of data objects in GBB, using lists as the object of study. The results indicate that high numbers of heterogeneous structures are present in GBB, though they are not highly heterogeneous: less than ten percent of the structures had more than 2 different internal element types. However, the sheer frequency of heterogeneity in GBB indicates a need to better accomodate this structure in future work on type flow analysis.

Chapter 6

Summary and Conclusions

This chapter concludes the dissertation, but not the research on type flow analysis for exploratory software development. It begins by summarizing the basic motivation and direction taken in the research presented in this thesis. From this basis, the next section of this chapter ties together the basic contributions of the research and provides an assessment of just how far type flow analysis takes us toward robust exploratory software. From this vantage point, the final section of the chapter discusses some particularly promising ways to build upon this research.

6.1 Type flow analysis: the status thus far

This research emerges from a long-standing attempt to understand why, behind closed doors, AI programmers are the object of friendly derision by software engineers, and vice-versa. On the one hand, the AI side delights in their ability to produce software behavior over a long weekend that might occupy the software engineers for an order of magnitude longer. On the other hand, software engineers love to attend AI software demonstrations, where

these systems blow up with clockwork regularity as soon as they encounter slightly unexpected demands.

The solution, as in most cultural disputes, seems to involve mutual respect. The AI community has justifiable pride in its exploratory programming languages and environments, which have produced substantial innovations in structuring techniques, software tools, and ways of viewing the development process. On the other hand, the software engineering community can justifiably worry about the means to these ends: if the price of these innovations is paid in software brittleness, then perhaps the price is too high.

Type flow analysis for exploratory software development is research on how to accomodate both of these positions. It began by examining what makes exploratory software exceptionally flexible and powerful yet simultaneously brittle, when compared with more traditional development techniques and languages. A constellation of answers emerged:

- The problems addressed by exploratory software—understanding natural language, designing human-computer interfaces, heuristic approaches to control of problem solving—are inherently underspecified problems. Traditional views of requirements and specifications cannot be successfully applied to these domains, yet these lifecycle phases are an important mechanism for removing brittleness from software.

- Exploratory software provides mechanisms for abstraction and genericity that provide significant support for the construction of systems. Exploratory languages allow program structures such as function and types to be defined at run-time, and make the types of all objects available for programmatic inspection at run-time. This allows very general

forms of polymorphism, which in turn supports the development of very abstract and general software structures.

- The type declarations that form the cornerstone of structural robustness in conventional software are inappropriate in exploratory software development.

  First, the underspecified nature of exploratory software induces significant structural evolution during the course of its development, and as noted in [She84], data types are a primary object of evolution. Maintaining appropriate type declarations can introduce significant developmental overhead.

  Second, the capabilities for polymorphism in exploratory software mean that conventional methods for type declaration, where the types of identifiers are declared once at their point of definition, are insufficient. In the limit, every *occurrence* of the identifier might need such annotation, and such declaration-based overhead is overwhelming even with small amounts of evolution.

- Parametric polymorphic type inference techniques, while solving the declaration overhead issue, do so by prohibiting language constructs employed in exploratory software system. Functional languages provide some support for polymorphism and higher-order programming, all without the need for type declarations. However, as the study of GBB in Chapter 5 illustrated, exploratory programs make significant use of ad-hoc polymorphism and heterogeneous data structures, which are prohibited in functional languages.

- The lack of type-level consistency assessment mechanisms is a primary

cause of brittleness in exploratory software systems. Both a study comparing the number of errors made by programmers using weakly and strongly typed languages [Gan77] and a study comparing the overall development process in a weakly and strongly typed language [Wol89] support this conclusion.

Providing some form of type consistency assessment in a manner compatible with both the language constructs and development process of exploratory software appeared to be a natural solution to the conundrum. The research then focussed on combining techniques from symbolic execution and polymorphic type inference to create an appropriate representation for type-level exploratory software structure. After a period of experimentation and evolution, a set of language constructs for the definition of this kind of type analysis system emerged, which became the basis for ESSIE. Finally, a particular instance of a type analysis system was implemented and used to assess the capabilities of the representation language on a small-scale exploratory system.

6.2 Contributions

This section sums up the basic contributions of type flow analysis for exploratory software development presented in this research. The contributions are summarized first, and then elaborated below.

Type flow analysis for exploratory software development:

1. Provides unique support for the *inference* of ad hoc polymorphism.

2. Supports broader forms of ad hoc polymorphism than other type assessment mechanisms.

3. Introduces a new representation for functional type.

4. Introduces novel mechanisms for the manipulation of this representation.

5. Supports the manipulation of *heterogeneous* as well homogeneous data objects.

6. Provides a novel programming language system, called ESSIE, for type analysis system construction.

7. Demonstrably provides unique support for the process of exploratory software development by finding type errors and improving the structural robustness of a small-scale exploratory system.

8. Represents important structural features of large scale exploratory software.

9. Uncovers principles of a "structured programming" method for exploratory software.

6.2.1 Representation and inference of ad hoc polymorphism

A key shortcoming of the polymorphic type inference techniques for functional languages is their limitation to a "particularly orderly form of polymorphism" [Car87] that represents type simply through a type expression for each parameter and result of a function. This parametric polymorphism is not even

sufficient to express the types of all predefined functions in their targetted languages (like ML): functions such as numeric operations are "hard-wired" with ad hoc polymorphic types.

Type classes in Haskell represent a particularly orderly form of *ad hoc* polymorphism: a form in which compile time information must suffice to translate an invocation of an ad-hoc function into an equivalent invocation of a parametric type, which is then analyzed by parametric inference. Type classes require the explicit declaration of all ad hoc functions.

Unlike parametric polymorphism, which prohibits ad hoc polymorphism entirely, and unlike type classes, which require the declaration of a restricted form of ad hoc polymorphism, type flow analysis is the first mechanism to actually *infer* the type of very general forms of ad hoc polymorphic functions. Type flow analysis does not require ad hoc functions to "resolve" to parametric functions during compilation: ambiguities in the functional type are represented and propagated throughout the system automatically.

To accomplish this, type flow analysis provides a new representation for functional type that blends control representations from symbolic execution systems with data representations from parametric polymorphism. Functions are not represented by simply assigning type expressions to parameters and results, but by representing the *type flow* through the function as a set of execution strands. Novel mechanisms for the construction, composition, and manipulation of this representation are developed in this research.

6.2.2 Representation of heterogeneous data objects

Another shortcoming of polymorphic type systems when applied to exploratory languages is their lack of support for heterogeneous data objects. In exploratory languages, structured objects like lists or hash tables can store arbitrarily different types of objects. In addition, variables can be assigned to different types of objects at different times. The type expression languages of ML or Haskell specifically exclude such heterogeneous data objects, which can also be viewed as ad hoc polymorphism at the *value* level.

The combination of ML-style type expressions and execution strand sets allows type flow analysis to support certain classes of heterogeneous data objects. Type flow analysis can model a variable that contains a list of both integers and strings by a set of two execution strands, one of which results in a list of integers and one of which results in a list of strings.[1] This representation is approximate, in that it does not preserve ordering of list elements, but it nonetheless provides a level of type representation and support for exploratory language constructs missing from other languages.

6.2.3 Design and implementation of a type analysis programming language

To test the ideas of type flow analysis, the ESSIE system was implemented. ESSIE does not simply implement a single type-level analysis system, but rather provides a programming language with which researchers can experiment with a broad class of type analysis systems and techniques.

[1]This same representation would be used to model a variable that contains *either* a list of integers *or* a list of strings, which exemplifies ad hoc polymorphism at the value level.

ESSIE's architecture is a contribution on two levels. First, by providing appropriate control and data abstractions, it facilitates experimentation. Second, these abstractions themselves provide insight into the fundamental nature of type flow analysis and polymorphic inference systems.

For example, the **define-analysis** data abstraction succinctly delineates the set of properties that must hold for an arbitrary term in ESSIE's type expression language (and, incidentally, in any parametric polymorphic type expression language).

With-strand and **with-arg-analyses** not only abstract away a great deal of computational complexity from the design of control flow analyzers, but they also circumscribe many of the fundamental extensions that type flow analysis makes to parametric polymorphic type inference. Within the scope of these abstractions, the developer can manipulate type expressions almost as if the elaborate execution strand representation did not exist.

6.2.4 Type flow analysis demonstrably supports software development

The techniques of type flow analysis were evaluated in part through the creation of a type analysis system using ESSIE, and its use in the analysis of a small scale exploratory software system. This analysis demonstrated that type flow analysis is able to provide support for the process of software development. Some examples of the support provided by ESSIE include the following:

- *The discovery of type-level errors.* ESSIE uncovered a previously unknown type-level error in the exploratory software under analysis. In

this situation, a function that should have tested to see if an argument was an instance of a type variable structure was actually testing to see if the argument was a symbol. ESSIE's analysis clearly pointed to the error, which was subsequently corrected.

- *The elucidation of subtle structural capabilities.* ESSIE's structural representation of functional type in some cases revealed previously unknown structural properties of functions. In one case, a function which was known to accept an argument of a list of type expressions was discovered by ESSIE to also accept an arbitrarily nested list of type expressions.

- *The elucidation of subtle structural brittleness.* The analysis made by ESSIE led to an unexpected result type for one of the functions in the exploratory system under analysis. Upon closer inspection, this result type was found to indicate a subtle structural brittleness in this function: while not technically incorrect, the function was prone to misuse if reused in new contexts. As a direct result of the analysis, the function was restructured into a more robust form.

To assess the relevence of the mechanisms of type flow analysis to "real world" exploratory software, a study of the type-level characteristics of a large scale exploratory system was undertaken. The results indicate that ad hoc polymorphism constitutes at least 40% of the polymorphism in the system, demonstrating the need for representations of the kind provided by type flow analysis.

6.2.5 Toward a "structured" style of exploratory programming

One of the triumphs of software engineering in the 1970's was the creation of structured programming techniques. This style of programming promotes more robust software by eschewing certain constructs, such as GOTO, which may be available in the implementation language, but whose use promotes structural features shown problematic for error prevention and software evolution.

Type flow analysis takes a few small steps toward the generation of a structured programming style for exploratory software development. Some of the programming tenets that arise from this research include:

- *Avoid gratuitous polymorphism.* Certain features in exploratory languages like Lisp introduce type-level ambiguity without providing significant expressive enhancements. For example, `nil` can be interpreted as a symbol or the empty list, while `null` can be interpreted as testing for either the symbol `nil` or an empty list. To use type flow analysis most effectively, such gratuitous polymorphism should be eschewed: use `(list)` rather than `nil` to indicate the empty list, and `endp` rather than `null` to test for an empty list.

- *GOTO's are still considered harmful.* Non-local flow of control constructs such as `return`, `catch`, and `tagbody` remain ill-advised in exploratory software development, just as constructs such as GOTO prove problematic in conventional development. However, while GOTO was castigated for its deleterious effect on *human* comprehension of program structure, `return` and its friends induce similar reactions from their effect on *Essie's* structural comprehension capabilities.

- *Avoid tricky conditionals.* Much of the precision of ESSIE's structural representations rest upon its ability to correctly interpret the type-level implications of conditional tests. For this reason, value-level computation should be performed in the body of conditionals rather than in conditional tests whenever possible. This edict prohibits the occasional use of conditional tests where value-level computation is "sandwiched in" for efficiency purposes.

  The "falling off the end of the cond clause" is another language feature shown suspect by ESSIE. In Lisp, if none of the conditional test clauses succeed, then the conditional returns `nil`. Such an interpretation of cond is normally not implemented in ESSIE's **cond** analyzer[2], since it often introduces additional, unintended ad hoc polymorphism into many functions. Plus, this interpretation is entirely superfluous: any cond clause that requires this behavior can explicitly indicate it through the addition of a final, `(t nil)` conditional arm.

- *Convert ordered heterogeneous data objects to globally defined structures.* ESSIE provides only partial support for heterogeneous objects, such that their representation only approximates their internal structure. For example, all representation of the order of internal elements is lost, so that a list of alternating integers and strings is represented as simply a list of either integers or strings, where operations that extract interior elements must always be prepared to accomodate either type of element. In many cases, this is acceptable for analysis purposes. However, greater represen-

[2]On the other hand, implementing this behavior in the analyzer could prove useful in trapping *unintentional* occurrences of falling off the end of the cond clause. Unfortunately, many cond clauses cannot be fallen off of because of their value-level characteristics, which might lead to ESSIE signalling ersatz errors.

tational precision can be provided by ESSIE if such heterogeneous data objects are converted to instances of the Lisp **structure** type, where ordering information can be explicitly declared.

6.2.6 Limitations of type flow analysis

Though the research direction pursued to date on type flow analysis has reaped many positive contributions, it has by no means completely solved the problem of developing robust exploratory software. In particular, several issues need to be addressed during the "scaling up" of type flow analysis to the real world:

- The current type analysis system developed using ESSIE does not implement all the type level objects and their relationships, control flow constructs, and predefined functions in Lisp.
- The current level of support for heterogeneous data objects in ESSIE may be too restrictive for many instances of exploratory software development. In particular, significant extensions to the parametric polymorphic type expression language adopted by ESSIE may be required to support this language feature.
- Precise characterization of the type level structure of exploratory software may sometimes require additional value-level representation beyond that currently implemented in ESSIE.

Even these shortcomings are ultimately encouraging, however, since they do not appear to indicate some fundamental flaw in the current approach to

type flow analysis, but rather the need for additional representational machinery beyond that which currently exists. The next section presents some ideas on addressing these issues, as well as new directions for exploration.

6.3 Future directions

6.3.1 Improved support for heterogeneity through fuzzy unification

One of the primary limitations of type flow analysis is its limited representation for heterogeneous data objects (although type flow analysis still provides more support for heterogeneity than other type inference mechanisms). Type flow analysis can exploit the representation of functions (and variables) as a set of execution strands to support the declaration of heterogeneous structures.

However, the type expression language employed in the current version of ESSIE is still fundamentally homogeneous in nature, which means that the type *inference* mechanism fails to support heterogeneity. This means that ESSIE requires user-supplied declarations in order to process forms such as:

```
(list 'a "a")
```

An important goal for future research is the provision of inference techniques for this rather common situation. Two paths immediately suggest themselves. The first path involves simply representing such a list as "a list whose first element is a symbol and whose second element is a string." Such a representation obviously wins in terms of precision, but poses significant difficulties when applied to the representation of recursive structures, the processing of loops, or the representation of infinite data objects (such as streams).

The second path sacrifices some precision for tractability: the list shown above would be inferred to be a list of either symbols or strings, without preservation of ordering information. This latter representation is closer to the declaration-based mechanisms currently provided, and is more amenable to the current inference techniques developed in this research.

Providing such an approximate inference facility in type flow analysis requires moving away from the conventional notion of unification toward "fuzzy unification," where the unification of two analysis objects results in a range of possible ways by which the two objects could be made equal, some of which might succeed, and some of which might fail. To give a flavor of processing with fuzzy unification, consider the following function:

```
(defun first-of-first (x)
  (car (car x)))
```

This function (popularly known as `caar`) accepts a list and takes the `car` of its first element. Now consider what happens when `first-of-first` is invoked with arguments of the following kinds of types:

- *A list of: lists of strings and lists of integers.* In this situation, `first-of-first` is guaranteed to return correctly: it's just not clear whether it will return a string or an integer, since ordering information is not preserved. In this case, fuzzy unification would have two successful outcomes, resulting in two execution strands, one for each possibility, being returned.

- *A list of: integers and lists of integers.* In this case, the invocation of `first-of-first` may or may not result in a type-level error, depending

upon whether the first element was an integer or a list. In this case, fuzzy unification would return both success and failure, and ESSIE would signal the potential for a type error in this situation.

- *A list of: integers and strings.* In this case, the invocation of `first-of-first` would definitely result in a type-level error regardless of the internal ordering of the list elements; all unification possibilities would fail, and ESSIE would signal a type error.

As the example suggests, such approximations of type level structure could result in a prohibitive number of "potential type error" messages during analysis. However, in combination with the judicious use of declarations, and the restructuring of heterogeneous data objects, such an inference capability may well prove valuable in the arsenal of tools for exploratory structure analysis.

6.3.2 Supporting the Common Lisp type hierarchy

The current type analysis system built with ESSIE supports type hierarchies constructed by the user through Lisp `structures`. Through use of the `:includes` option to `defstruct` (and to `define-unit`), a hierarchy of structures that share slots and slot accessor functions can be defined.

However, Lisp predefined types can also be organized into a hierarchy based upon the presence of predefined functions that operate across a spectrum of classes. For example, various Lisp functions such as `length` and `subseq` accept objects of type `sequence`, which encompasses both lists and vectors. Vectors, in turn, encompass both one dimensional arrays as well as strings, and so forth. The analyzers and analysis objects implemented for this dissertation do not

model the Common Lisp type hierarchy, but instead model a flat cross-section of a subset of these types.

While the currently implemented type analysis system does not support non-structured declared hierarchies, ESSIE was designed with the goal of providing this functionality. Certain design aspects of the type flow analysis language that may have appeared unorthodox become more understandable in this light.

For example, the arguments passed to the *unifier* function in **define-analysis** may be not only objects of the analysis type being defined, but any other pair of analysis objects as well. In the flat type system currently defined, this leads to analysis objects immediately testing to make sure the arguments are of the analysis type under definition, with the *unifier* function failing unless this precondition is met. In a hierarchical type system, a unifier function would test to see if the arguments are of the type of the current analysis object being defined, or of any *supertype* of that type. If the object was a supertype, the function would instantiate the object to its subtype before proceeding with the unification of interior structural elements.

6.3.3 Providing type-level information to the underlying environment

As discussed in Chapter 4, Common Lisp provides facilities such as `declare` and `the` for the declaration of type-level information. These facilities are specifically excluded from use in type consistency assessment, since implementations of Common Lisp are free to ignore the type-level information they provide. Rather, they are intended as a signal to the Common Lisp compiler

that optimizations may be possible on the object code within the scope of the declaration. The work by Beer [Bee88] illustrates the potential benefits from providing this information.

Type flow analysis provides a powerful framework for the provision of this information. An interesting extension of the current work involves letting ESSIE insert declarations around forms specifying their return value types, and seeing if this information suffices for object code optimzation. Unfortunately, facilities such as the allow declaration of only a single result type, which limits its utility to single-stranded braids, or multiple stranded braids with identical result types. This limitation might be overcome by appropriate use of the `deftype` type specification facility, however.

6.3.4 Supporting the Common Lisp Object System

The Common Lisp Object System (CLOS) [BDG+88] is the standard object oriented extension facility to Common Lisp. CLOS provides the *class* construct to encapsulate data and procedures, and *generic functions* that are composed from *methods* to support the inheritance and composition of behavior.

The ability of type flow analysis to exploit type predicates in the process of understanding the type-level structure of functions provides promise for its extension to CLOS. This results from the fact that the generic function facility that forms the basis of CLOS operates much like the generation of a type predicate-based conditional.

Consider the example CLOS mechanism for defining a function `area` that applies to a variety of different shapes, as shown in Figure 42. First, the

• *The CLOS definition for a generic* area *function:*

```
(defgeneric area (shape))

(defmethod area ((shape circle))
    ...)
(defmethod area ((shape rectangle))
    ...)
(defmethod area ((shape triangle))
   ...)
```

• *The type predicate-based model of the* area *generic function:*

```
(defun area (shape)
  (cond ((circle-p shape)
         ...)
        ((rectangle-p shape)
         ...)
        ((triangle- shape)
         ...)))
```

Figure 42: The CLOS and type predicate versions of AREA

generic function area is defined, and then methods for supporting the area computation on various kinds of shapes (circles, rectangles, and triangles) are specified. (Class definitions for these kinds of objects would be required but are not illustrated here.)

The implementation of the generic function area built from this simple object oriented definition can be straightforwardly modeled by a definition based upon type predicates, as shown at the bottom of Figure 42. The structure of this latter version of area is easily analyzed by type flow analysis.

The correspondence between CLOS generic functions and type predicate-based dispatching functions raises hopes that type flow analysis can be extended to the analysis of object oriented systems. Such explorations form an exciting future direction for this line of research.

6.4 Epilogue

All good research can be cast as a quest for some holy grail, and this research shares that quality.[3] In this case, the holy grail is the "ideal" programming language and environment: one that offers unlimited rope, and allows that rope to be easily tied and untied in all manner of ways, yet steadfastly prevents that rope from being fashioned into a noose.

As usual, this quest has fallen short of its hoped for goal. Type flow analysis does not singlehandedly "solve" the problem of brittle exploratory software. It does, however, implement an approach—one based upon respect for the fundamental premises of both the AI and SE communities, but one that simultaneously challenges the status quo in both communities.

Type flow analysis challenges the AI/exploratory software community to focus more attention on the structural robustness of their systems. As tools for type-level asssessment like the one presented in this research mature, this community will no longer be able to excuse lazy development practices by facile manipulation of the run-time stack from within the debugger. They will need to restructure software that *runs* into software that can be verified to *run*

[3] And perhaps that quality alone.

without type-level errors; they may even need to provide an occasional type declaration.

Type flow analysis challenges the software engineering community to take exploratory languages and development practices seriously, as a source of ideas and even wisdom about software structuring techniques. This research suggests that ad hoc polymorphism is not "ad hoc" at all, but rather a general purpose, widely applicable, and powerful structuring facility that does not preclude the goal of type-level consistency. Finally, it urges the SE community to abandon the use of strong typing as a litmus test for "serious" programming languages, and to view it instead as an important property that individual software *systems* should strive for over the course of their lifetimes.

APPENDIX A

PTI: A PARAMETRIC POLYMORPHIC TYPE INFERENCE SYSTEM

This appendix section lists the source code for PTI and the analysis provided for it by ESSIE. PTI is a Lisp implementation of parametric polymorphic type inference following the example in [Car87].

Following each definition of a PTI function is its analysis in "display" format.

```
;; -*- Mode:Common-Lisp; Package:PTI; Fonts:(CPTFONT); Base:10 -*-
;; *-* File: Deputy:CER$DISK:[JOHNSON.INFER.PTI]PTI-ANNOTATED.LISP *-*
;; *-* Last-Edit: Thursday, June 28, 1990  20:32:19; Edited-By: JOHNSON *-*

;;; ------------------------------------------------------------------------
;;; PTI structure definitions
;;; ------------------------------------------------------------------------

(defstruct (t-var (:conc-name "T-VAR.")
                  (:constructor make-t-var (instance)))
  "Type variables are instantiated by setting their INSTANCE field."
  instance)

;Function MAKE-T-VAR
;<(INSTANCE Nil)> ==>> T-Var
;<(INSTANCE T-Var)> ==>> T-Var
;<(INSTANCE T-Op)> ==>> T-Var


;Function T-VAR.INSTANCE
;<(T-VAR T-Var)> ==>> Nil
;<(T-VAR T-Var)> ==>> T-Var
;<(T-VAR T-Var)> ==>> T-Op
```

```
;;; ---------------------------------------------------------------------

(defstruct (t-op (:conc-name "T-OP.")
                 (:constructor make-t-op (name args)))
  "Type operators have a name (such as BOOLEAN, LIST, or FUNCTION), and
   a list of type expressions."
  name
  args)

;Function MAKE-T-OP
;<(ARGS LIST[T-Op)(NAME Sym)> ==>> T-Op
;<(ARGS LIST[T-Var])(NAME Sym)> ==>> T-Op
;<(ARGS Nil)(NAME Sym)> ==>> T-Op

;Function T-OP.NAME
;<(T-OP T-Op)> ==>> Sym

;Function T-OP.ARGS
;<(T-OP T-Op)> ==>> LIST[T-Op]
;<(T-OP T-Op)> ==>> LIST[T-Var]
;<(T-OP T-Op)> ==>> Nil

;;; ---------------------------------------------------------------------

(defstruct (env (:conc-name "ENV.")
                (:constructor make-env (identifier t-op)))
  "Environment structures bind an identifier to a type expression."
  identifier
  t-op)

;Function MAKE-ENV
;<(T-OP T-Var)(IDENTIFIER Sym)> ==>> Env
;<(T-OP T-Op)(IDENTIFIER Sym)> ==>> Env

;Function ENV.T-OP
;<(ENV Env)> ==>> T-Var
;<(ENV Env)> ==>> T-Op

;Function ENV.IDENTIFIER
;<(ENV Env)> ==>> Sym

;;; ---------------------------------------------------------------------
;;; Definition and operations on atomic (base) types
;;; ---------------------------------------------------------------------

(defparameter *atomic-type-table* (make-hash-table)
  "A hash table containing base type expressions.")
```

```
;Function *ATOMIC-TYPE-TABLE*
; <Empty> ==>> Table<Sym><T-Op>

;;; ---------------------------------------------------------------------

(defun add-atomic-type (identifier)
  "Adds an atomic type"
  (setf (gethash identifier *atomic-type-table*)
        (make-t-op identifier (list))))

;Function ADD-ATOMIC-TYPE
;<(IDENTIFIER Sym) > ==>> T-Op

;;; ---------------------------------------------------------------------

(defun setup-atomic-types ()
  "Initializes the atomic type table"
  (mapc #'add-atomic-type '(boolean symbol number string)))

;Function SETUP-ATOMIC-TYPES
; <Empty> ==>> Sym

;;; ---------------------------------------------------------------------

(defun get-atomic-type (name)
  "Retrieves an atomic type"
  (gethash name *atomic-type-table*))

;Function GET-ATOMIC-TYPE
;<(NAME Sym) > ==>> T-Op

;;; ---------------------------------------------------------------------
;;; The Global Environment
;;; ---------------------------------------------------------------------

(defparameter *global-env* nil
  "A list of ENV structures representing predefined functions.")

;Function *GLOBAL-ENV*
; <Empty> ==>> LIST[Env]

;;; ---------------------------------------------------------------------


(defun add-global-binding (identifier type-name arg-list)
  "Adds IDENTIFIER and its type to the global environment."
  (push (make-env identifier (make-t-op type-name arg-list)) *global-env*))
```

```
;Function ADD-GLOBAL-BINDING
;<(IDENTIFIER Sym)(TYPE-NAME Sym)(ARG-LIST LIST[T-Op])> ==>>LIST[Env]
;<(IDENTIFIER Sym)(TYPE-NAME Sym)(ARG-LIST LIST[T-Var]) > ==>> LIST[Env]
;<(IDENTIFIER Sym)(TYPE-NAME Sym)(ARG-LIST Nil) > ==>> LIST[Env]

;;; ---------------------------------------------------------------------

(defun get-global-binding (identifier env)
  "Retrieves the binding associated with IDENTIFIER from ENV.
   Returns NIL if not found."
  (let ((env-struct (find identifier env :key #'env.identifier)))
    (if env-struct
        (env.t-op env-struct)
        nil)))

;Function GET-GLOBAL-BINDING
;<(IDENTIFIER Sym)(ENV LIST[Env]) > ==>> T-Var
;<(IDENTIFIER Sym)(ENV LIST[Env]) > ==>> T-Op
;<(IDENTIFIER TV-9)(ENV LIST[Env]) > ==>> Nil
;<(IDENTIFIER Sym)(ENV LIST[Env]) > ==>> Nil

;;; ---------------------------------------------------------------------

(defun setup ()
  (setup-atomic-types)
  (add-global-binding '+ 'function
                      (list (get-atomic-type 'number)
                            (get-atomic-type 'number)))

  (add-global-binding 'null 'function
                      (list (make-t-op 'list (list (make-t-var nil)))
                            (get-atomic-type 'boolean)))

  (add-global-binding 'cdr 'function
                      (let ((list-var (make-t-op 'list
                                                 (list (make-t-var nil)))))
                        (list list-var list-var)))

  (add-global-binding 'car 'function
                      (let* ((var (make-t-var nil))
                             (list-type (make-t-op 'list (list var))))
                        (list list-type var)))

  (add-global-binding 'quote 'function
                      (let ((list-var (make-t-op 'list
                                                 (list (make-t-var nil)))))
                        (list list-var list-var)))
```

```
  (add-global-binding 'succ 'function
                      (list (get-atomic-type 'number)
                            (get-atomic-type 'number))))

;Function SETUP
; <Empty> ==>> LIST[Env]

;;; ------------------------------------------------------------------------
;;; ANALYZE-EXP and ANALYZE-FORM
;;; ------------------------------------------------------------------------

(defun analyze-form (form)
  (analyze-exp form *global-env* (list)))

;Function ANALYZE-FORM
;<(FORM LIST[TV-28]) > ==>> T-Var
;<(FORM LIST[TV-32]) > ==>> T-Op
;<(FORM Str) > ==>> T-Op
;<(FORM Int) > ==>> T-Op
;<(FORM Sym) > ==>> T-Op

;;; ------------------------------------------------------------------------

(defun analyze-exp (exp env nongenvars)
  "Given an expression (EXP), an environment of variable bindings (ENV),
   and a list of the current nongeneric variables (NONGENVARS),
   ANALYZE-EXP returns a type expression for EXP."

  (cond

    ((if-p exp)
     (analyze-if exp env nongenvars))

    ((defun-p exp)
     (analyze-defun exp env nongenvars))

    ((let-p exp)
     (analyze-let exp env nongenvars))

    ((listp exp)
     (analyze-application exp env nongenvars))

    ((funcall-p exp)
     (analyze-funcall exp env nongenvars))

    ((and (symbolp exp) (retrieve-type exp env nongenvars)))
```

```
    (t
     (retrieve-primitive-type exp))))

;Function ANALYZE-EXP
;<(EXP LIST[TV-48])(ENV LIST[Env])(NONGENVARS LIST[T-Var])> ==>>T-Var
;<(EXP LIST[TV-52])(ENV LIST[Env])(NONGENVARS LIST[T-Var])> ==>>T-Op
;<(EXP Str)(ENV LIST[Env])(NONGENVARS LIST[T-Var])> ==>> T-Op
;<(EXP Int)(ENV LIST[Env])(NONGENVARS LIST[T-Var])> ==>> T-Op
;<(EXP Sym)(ENV LIST[Env])(NONGENVARS LIST[T-Var])> ==>> T-Op

;;; ------------------------------------------------------------------------
;;; Lisp form identification predicates
;;; ------------------------------------------------------------------------

(defun if-p (exp)
  "If EXP is a if-clause, returns T, otherwise NIL."
  (and (listp exp)
       (eq (first exp) 'IF)))

;Function IF-P
;<(EXP LIST[Sym])> ==>> Boolean[T]
;<(EXP TV-8)> ==>> Boolean[F]
;   Where: (NOT (LISTP TV-8))
;<(EXP LIST[TV-12])> ==>> Boolean[F]
;<(EXP LIST[Sym])> ==>> Boolean[F]

;;; ------------------------------------------------------------------------

(defun defun-p (exp)
  "If EXP is a defun form, returns T, otherwise NIL."
  (and (listp exp)
       (eq (first exp) 'DEFUN)))

;Function DEFUN-P
;<(EXP LIST[Sym])> ==>> Boolean[T]
;<(EXP TV-8)> ==>> Boolean[F]
;   Where: (NOT (LISTP TV-8))
;<(EXP LIST[TV-12])> ==>> Boolean[F]
;<(EXP LIST[Sym])> ==>> Boolean[F]

;;; ------------------------------------------------------------------------


(defun let-p (exp)
  "If EXP is a let form, returns T, otherwise NIL."
  (and (listp exp)
       (eq (first exp) 'LET)))
```

```
;Function LET-P
;<(EXP LIST[Sym])> ==>> Boolean[T]
;<(EXP TV-8)> ==>> Boolean[F]
;   Where: (NOT (LISTP TV-8))
;<(EXP LIST[TV-12])> ==>> Boolean[F]
;<(EXP LIST[Sym])> ==>> Boolean[F]

;;; ------------------------------------------------------------------------

(defun funcall-p (exp)
  "If EXP is a funcall form, returns T, otherwise NIL."
  (and (listp exp)
       (eq (first exp) 'FUNCALL)))

;Function FUNCALL-P
;<(EXP LIST[Sym])> ==>> Boolean[T]
;<(EXP TV-8)> ==>> Boolean[F]
;   Where: (NOT (LISTP TV-8))
;<(EXP LIST[TV-12])> ==>> Boolean[F]
;<(EXP LIST[Sym])> ==>> Boolean[F]


;;; ------------------------------------------------------------------------
;;; Lisp Form Type Analysis Functions
;;; ------------------------------------------------------------------------

(defun analyze-if (if-form env nongenvars)
  "Returns a type expression for IF-FORM"
  (let ((test (analyze-exp (second if-form) env nongenvars))
        (then (analyze-exp (third if-form) env nongenvars))
        (else (analyze-exp (fourth if-form) env nongenvars)))
    (declare (ignore test))
    (unify-type then else)
    else))

;Function ANALYZE-IF
;<(IF-FORM LIST[Sym])(ENV LIST[Env])(NONGENVARS LIST[T-Var])>==>> T-Op
;<(IF-FORM LIST[Int])(ENV LIST[Env])(NONGENVARS LIST[T-Var])>==>> T-Op
;<(IF-FORM LIST[Str])(ENV LIST[Env])(NONGENVARS LIST[T-Var])> ==>> T-Op
;<(IF-FORM LIST[LIST[TV-68]])(ENV LIST[Env])(NONGENVARS LIST[T-Var]) >
;      ==>> T-Op
;<(IF-FORM LIST[LIST[TV-72]])(ENV LIST[Env])(NONGENVARS LIST[T-Var]) >
;      ==>> T-Var

;;; ------------------------------------------------------------------------


(defun analyze-defun (defun-form env nongenvars)
```

```
  "Returns a type expression for a defun form."
  (let* ((new-ident-list (list (second defun-form) (third defun-form)))
         (new-t-vars (mapcar #'(lambda (param)
                                 (declare (ignore param))
                                 (make-t-var nil))
                             new-ident-list))
         (new-env (extend-env new-ident-list new-t-vars env))
         (new-nongenvars (extend-nongenvars new-t-vars nongenvars))
         (result-type (analyze-progn (cdddr defun-form) new-env
                                     new-nongenvars)))

    (unify-type (make-t-op 'function
                           (append (cdr new-t-vars) (list result-type)))
                (retrieve-type (second defun-form) new-env new-nongenvars))
    (retrieve-type (second defun-form) new-env new-nongenvars)))

;Function ANALYZE-DEFUN
;<(DEFUN-FORM LIST[Sym])(ENV LIST[Env])(NONGENVARS LIST[T-Var]) >
;   ==>> T-Var
;<(DEFUN-FORM LIST[Sym])(ENV LIST[Env])(NONGENVARS LIST[T-Var]) >
;   ==>> T-Op
;<(DEFUN-FORM LIST[Sym])(ENV LIST[Env])(NONGENVARS LIST[T-Var]) >
;   ==>> Boolean[F]

;;; ---------------------------------------------------------------------

(defun analyze-let (let-form env nongenvars)
  "Returns a type expression for LET."
  ;;first we augment ENV to include the bindings in the let forms.

  (let* ((bindings (second let-form))
         (let-body (cddr let-form))
         (new-env (extend-env
                    (mapcar #'(lambda (var-value-pair)
                                (first var-value-pair))
                            bindings)
                    (mapcar #'(lambda (var-value-pair)
                                (analyze-exp (second var-value-pair)
                                             env nongenvars))
                            bindings)
                    env)))

    ;;now that NEW-ENV has the right bindings, analyze the let body
    ;;the type of the last form is the type of the entire let.
    (analyze-progn let-body new-env nongenvars)))

;Function ANALYZE-LET
;<(LET-FORM LIST[LIST[LIST[Sym]]])(ENV LIST[Env])(NONGENVARS LIST[T-Var])>
```

```
;   ==>> T-Op
;<(LET-FORM LIST[LIST[LIST[Sym]]])(ENV LIST[Env])(NONGENVARS LIST[T-Var])>
;   ==>> T-Var
;<(LET-FORM LIST[LIST[LIST[Int]]])(ENV LIST[Env])(NONGENVARS LIST[T-Var])>
;   ==>> T-Op
;<(LET-FORM LIST[LIST[LIST[Int]]])(ENV LIST[Env])(NONGENVARS LIST[T-Var])>
;   ==>> T-Var
;<(LET-FORM LIST[LIST[LIST[Str]]])(ENV LIST[Env])(NONGENVARS LIST[T-Var])>
;   ==>> T-Op
;<(LET-FORM LIST[LIST[LIST[Str]]])(ENV LIST[Env])(NONGENVARS LIST[T-Var])>
;   ==>> T-Var
;<(LET-FORM LIST[LIST[LIST[LIST[TV-176]]]])(ENV LIST[Env])
;   (NONGENVARS LIST[T-Var])> ==>> T-Op
;<(LET-FORM LIST[LIST[LIST[LIST[TV-184]]]])(ENV LIST[Env])
;   (NONGENVARS LIST[T-Var])> ==>> T-Var

;;; ------------------------------------------------------------------------

(defun analyze-application (appl env nongenvars)
  "Return a type expression for APPL (i.e. a regular old function call.)"

  (let* ((name (car appl))
         (arg-list (cdr appl))
         (function-type (analyze-exp name env nongenvars))
         (arg-list-types (mapcar #'(lambda (arg)
                                     (analyze-exp arg env nongenvars))
                                 arg-list))
         (result-type (make-t-var nil)))
    (unify-type function-type
                (make-t-op 'function
                           (append arg-list-types (list result-type))))

    ;;original call ommitted prune.
    (prune result-type)))

;Function ANALYZE-APPLICATION
;<(APPL LIST[LIST[TV-40]])(ENV LIST[Env])(NONGENVARS LIST[T-Var]) >
;   ==>> T-Op
;<(APPL LIST[LIST[TV-44]])(ENV LIST[Env])(NONGENVARS LIST[T-Var]) >
;   ==>> T-Var
;<(APPL LIST[Sym])(ENV LIST[Env])(NONGENVARS LIST[T-Var]) >
;   ==>> T-Op
;<(APPL LIST[Sym])(ENV LIST[Env])(NONGENVARS LIST[T-Var]) >
;   ==>> T-Var

;;; ------------------------------------------------------------------------
```

```
(defun analyze-funcall (funcall-form env nongenvars)
  "Return a type expression for a funcall.  Since our little parser
   doesn't keep separate function and value cells, we can pretend
   that this is Scheme for the moment."
  (analyze-application (cdr funcall-form) env nongenvars))

;Function ANALYZE-FUNCALL
;<(FUNCALL-FORM LIST[LIST[TV-40]])(ENV LIST[Env])(NONGENVARS LIST[T-Var])>
;   ==>> T-Op
;<(FUNCALL-FORM LIST[LIST[TV-44]])(ENV LIST[Env])(NONGENVARS LIST[T-Var])>
;   ==>> T-Var
;<(FUNCALL-FORM LIST[Sym])(ENV LIST[Env])(NONGENVARS LIST[T-Var]) >
;   ==>> T-Op
;<(FUNCALL-FORM LIST[Sym])(ENV LIST[Env])(NONGENVARS LIST[T-Var]) >
;   ==>> T-Var

;;; ----------------------------------------------------------------------


(defun analyze-progn (progn-form env nongenvars)
  "Return a type expression for a PROGN form."
  ;;analyze the expressions in PROGN-FORM, returning the last one's type.

  (cond ((= 1 (length progn-form))
         (analyze-exp (first progn-form) env nongenvars))
         (t
           (analyze-exp (first progn-form) env nongenvars)
           (analyze-progn (cdr progn-form) env nongenvars))))

;Function ANALYZE-PROGN
;<(PROGN-FORM LIST[Sym])(ENV LIST[Env])(NONGENVARS LIST[T-Var])>
;   ==>> T-Op
;<(PROGN-FORM LIST[Int])(ENV LIST[Env])(NONGENVARS LIST[T-Var])>
;   ==>> T-Op
;<(PROGN-FORM LIST[Str])(ENV LIST[Env])(NONGENVARS LIST[T-Var])>
;   ==>> T-Op
;<(PROGN-FORM LIST[LIST[TV-76]])(ENV LIST[Env])(NONGENVARS LIST[T-Var]>
;   ==>> T-Op
;<(PROGN-FORM LIST[LIST[TV-81]])(ENV LIST[Env])(NONGENVARS LIST[T-Var])>
;   ==>> T-Var


;;; ----------------------------------------------------------------------
;;; 'Extend' routines for environments and nongeneric variable lists.
;;; ----------------------------------------------------------------------

(defun extend-env (identifiers types env)
  "Return a new list of environment structures with IDENTIFIERS bound
```

```
   to their corresponding TYPES.
   IDENTIFIERS and TYPES may be lists or atoms."
  (cond ((and (not (listp identifiers)) (not (listp types)))
         (cons (make-env identifiers types) env))

        ((and (listp identifiers) (listp types))
         (append (mapcar #'make-env identifiers types) env))))

;Function EXTEND-ENV
;<(IDENTIFIERS Sym)(TYPES T-Op)(ENV LIST[Env])> ==>> LIST[Env]
;<(IDENTIFIERS Sym)(TYPES T-Var)(ENV LIST[Env])> ==>> LIST[Env]
;<(IDENTIFIERS LIST[Sym])(TYPES LIST[T-Op])(ENV LIST[Env]) >
;   ==>> LIST[Env]
;<(IDENTIFIERS LIST[Sym])(TYPES LIST[T-Var])(ENV LIST[Env]) >
;   ==>> LIST[Env]

;;; ------------------------------------------------------------------------

(defun extend-nongenvars (vars nongenvars)
  "Return a new nongenvars list with vars (which may be a symbol or list)."
  (if (t-var-p vars)    ;; originally incorrect: (symbolp vars)
      (cons vars nongenvars)
      (append vars nongenvars)))

;Function EXTEND-NONGENVARS
;<(VARS T-Var)(NONGENVARS LIST[T-Var])> ==>> LIST[T-Var]
;<(VARS LIST[TV-5])(NONGENVARS LIST[TV-5])> ==>> LIST[TV-5]

;;; ------------------------------------------------------------------------


(defun prune (type-exp)
  "Returns a 'pruned' version of TYPE-EXP"
  (cond
    ((t-op-p type-exp)
     type-exp)

    ((and (t-var-p type-exp) (null (t-var.instance type-exp)))
     type-exp)

    ((t-var-p type-exp)
     (prune (t-var.instance type-exp)))))

;Function PRUNE
;<(TYPE-EXP T-Op)> ==>> T-Op
;<(TYPE-EXP T-Var)> ==>> T-Var
;<(TYPE-EXP T-Var)> ==>> T-Op
;<(TYPE-EXP T-Var)> ==>> T-Var
```

```
;;; ------------------------------------------------------------------------
;;; Occurs-check functions for unification routine.
;;; ------------------------------------------------------------------------


(defun occurs-in-p (t-var t-exps)
  "Check to see if T-VAR occurs in T-EXPS."
  (let ((t-var (prune t-var))
        (t-exps (if (listp t-exps)
                    t-exps
                    (prune t-exps))))

    (cond ((t-var-p t-var)
           (cond ((t-var-p t-exps)
                  (eq t-var t-exps))

                 ((and (listp t-exps) (endp t-exps))
                  'nil)

                 ((t-op-p t-exps)
                  (occurs-in-p t-var (t-op.args t-exps)))

                 ((listp t-exps)
                  (if (occurs-in-p t-var (first t-exps))
                      'T
                      (occurs-in-p t-var (rest t-exps)))))))))

;Function OCCURS-IN-P
;<(T-VAR T-Var)(T-EXPS T-Op)> ==>> Boolean[T]
;<(T-VAR T-Var)(T-EXPS T-Op)> ==>> Boolean[F]
;<(T-VAR T-Var)(T-EXPS T-Var)> ==>> Boolean[T]
;<(T-VAR T-Var)(T-EXPS T-Var)> ==>> Boolean[F]
;<(T-VAR T-Var)(T-EXPS LIST[T-Op])> ==>> Boolean[T]
;<(T-VAR T-Var)(T-EXPS LIST[T-Op])> ==>> Boolean[F]
;<(T-VAR T-Var)(T-EXPS LIST[T-Var])> ==>> Boolean[T]
;<(T-VAR T-Var)(T-EXPS LIST[T-Var])> ==>> Boolean[F]



;;; ------------------------------------------------------------------------
;;; Unification functions.
;;; ------------------------------------------------------------------------

(defun unify-type (type-exp1 type-exp2)
  "Unify two type expressions together."
  ;; first, prune the type expressions.
  (let ((type-exp1 (prune type-exp1))
```

```
        (type-exp2 (prune type-exp2)))

    (cond ((t-var-p type-exp1)
           ;;exp1=(uninstantiated) t-var => 'replace it' by exp2.
           ;;but check for circularity, first.
           (cond ((eq type-exp1 type-exp2)
                  nil) ;;don't do anything---they're equal.

                 ((occurs-in-p type-exp1 type-exp2)
                  (error "Famous unification 'occurs check' problem."))

                 (t
                   (setf (t-var.instance type-exp1) type-exp2)
                  nil)))

          ((and (t-op-p type-exp1) (t-var-p type-exp2))
           ;;'retry' unification with reversed args.
           (unify-type type-exp2 type-exp1))

          ((and (t-op-p type-exp1) (t-op-p type-exp2))
           ;;unify two 'compound' type structures.
           (if (eq (t-op.name type-exp1) (t-op.name type-exp2))
               (unify-t-op-arg-list type-exp1 type-exp2)
               (error "Type clash between:~%~s~%~s
                      ~%~a doesn't match ~a" type-exp1 type-exp2
                      (t-op.name type-exp1) (t-op.name type-exp2)))))))

;Function UNIFY-TYPE
;<(TYPE-EXP1 T-Var)(TYPE-EXP2 T-Var)> ==>> Nil
;<(TYPE-EXP1 T-Var)(TYPE-EXP2 T-Op)> ==>> Nil
;<(TYPE-EXP1 T-Op)(TYPE-EXP2 T-Op)> ==>> Nil
;<(TYPE-EXP1 T-Op)(TYPE-EXP2 T-Var)> ==>> Nil
;<(TYPE-EXP1 T-Var)(TYPE-EXP2 T-Var)> ==>> Nil

;;; ---------------------------------------------------------------------

(defun unify-t-op-p (type-exp1 type-exp2)
  "Unify the argument lists of TYPE-EXP1 to TYPE-EXP2"
  ;;call unify-type on corresponding elements of each arglist.
  ;;but make sure they're 'structurally compatible.'

  (let ((arglist1 (t-op.args type-exp1))
        (arglist2 (t-op.args type-exp2)))

    (if (= (length arglist1) (length arglist2))
        (mapc #'unify-type arglist1 arglist2)
        (error "Type clash between: ~%~s (~d args)~%~s (~d args):"
               type-exp1 (length arglist1) type-exp2 (length arglist2)))))
```

```
;Function UNIFY-T-OP-P
;<(TYPE-EXP1 T-Op)(TYPE-EXP2 T-Op)> ==>> Nil)

;;; ---------------------------------------------------------------------

(defun generic-var-p (t-var nongenvars)
  "Returns T if T-VAR does not occur in NONGENVARS,
   NIL otherwise."
  (not (occurs-in-p t-var nongenvars)))

;Function GENERIC-VAR-P
;<(T-VAR T-Var)(NONGENVARS LIST[T-Var])> ==>> Boolean[T]
;<(T-VAR T-Var)(NONGENVARS LIST[T-Var])> ==>> Boolean[F]

;;; ---------------------------------------------------------------------
;;; Type expression creation machinery
;;; ---------------------------------------------------------------------

(defun fresh-type (type-exp nongenvars)
  "Returns a copy of TYPE-EXP, with generic variables duplicated."

  (let ((*generic-var-list* (list)))
    (declare (special *generic-var-list*))
    ;; *generic-var-list* is an alist of the duplicated generic vars
    ;; and their replacements.  It accumulates generic-var replacements.
    (fresh type-exp nongenvars)))

;Function FRESH-TYPE
;<(TYPE-EXP T-Op)(NONGENVARS LIST[T-Var])> ==>> T-Op
;<(TYPE-EXP T-Var)(NONGENVARS LIST[T-Var])> ==>> T-Var
;<(TYPE-EXP T-Var)(NONGENVARS LIST[T-Var])> ==>> T-Op
;<(TYPE-EXP T-Var)(NONGENVARS LIST[T-Var])> ==>> T-Var

;;; ---------------------------------------------------------------------

(defun fresh (type-exp nongenvars)
  "Creates a fresh instance of TYPE-EXP."

  (let ((type-exp (prune type-exp)))
    (cond
      ((t-var-p type-exp)
       (if (generic-var-p type-exp nongenvars)
           (fresh-var type-exp)
           type-exp))

      ((t-op-p type-exp)
       (make-t-op (t-op.name type-exp)
```

```
                (fresh-list (t-op.args type-exp) nongenvars))))))

;Function FRESH
;<(TYPE-EXP T-Op)(NONGENVARS LIST[T-Var])> ==>> T-Op
;<(TYPE-EXP T-Var)(NONGENVARS LIST[T-Var])> ==>> T-Var
;<(TYPE-EXP T-Var)(NONGENVARS LIST[T-Var])> ==>> T-Op
;<(TYPE-EXP T-Var)(NONGENVARS LIST[T-Var])> ==>> T-Var

;;; ---------------------------------------------------------------------

(defun fresh-list (type-list nongenvars)
  "Make a 'fresh' copy of the list of arguments to a type operator."
  (mapcar #'(lambda (type-exp)
              (fresh type-exp nongenvars))
          type-list))

;Function FRESH-LIST
;<(TYPE-LIST Nil)(NONGENVARS LIST[T-Var])> ==>> Nil
;<(TYPE-LIST LIST[T-Var])(NONGENVARS LIST[T-Var])> ==>> T-Var
;<(TYPE-LIST LIST[T-Var])(NONGENVARS LIST[T-Var])> ==>> T-Op
;<(TYPE-LIST LIST[T-Op])(NONGENVARS LIST[T-Var])> ==>> T-Var
;<(TYPE-LIST LIST[T-Op])(NONGENVARS LIST[T-Var])> ==>> T-Op

;;; ---------------------------------------------------------------------

(defun fresh-var (t-var)
  "Make a 'fresh' copy of the generic type variable T-VAR."
  ;;If we have a fresh version of T-VAR already, use it.

  (declare (special *generic-var-list*))
  (let* ((var-pair (find t-var *generic-var-list* :key #'car) )
         (fresh-var (if var-pair (second var-pair))))

    (cond (fresh-var
           fresh-var)
          (t
           (let ((new-var (make-t-var nil)))
             (push (list t-var new-var) *generic-var-list*)
             new-var)))))

;Function FRESH-VAR
;<(T-VAR T-Var)> ==>> T-Var

;;; ---------------------------------------------------------------------
;;; Retrieval functions for identifiers and objects.
;;; ---------------------------------------------------------------------

(defun retrieve-type (identifier env nongenvars)
```

```
  "Returns the 'fresh' type of IDENTIFIER.
   Returns NIL if IDENTIFIER is not found in ENV."

  (let ((identifier-type-pair (get-global-binding identifier env)))
    (and identifier-type-pair
         (fresh-type identifier-type-pair nongenvars))))

;Function RETRIEVE-TYPE
;<(IDENTIFIER Sym)(ENV LIST[Env])(NONGENVARS LIST[T-Var])> ==>> T-Var
;<(IDENTIFIER Sym)(ENV LIST[Env])(NONGENVARS LIST[T-Var])> ==>> T-Op
;<(IDENTIFIER Sym)(ENV LIST[Env])(NONGENVARS LIST[T-Var])> ==>> Boolean[F]
;<(IDENTIFIER TV-19)(ENV LIST[Env])(NONGENVARS TV-22)> ==>> Boolean[F]
;<(IDENTIFIER Sym)(ENV LIST[Env])(NONGENVARS TV-27)> ==>> Boolean[F]

;;; ------------------------------------------------------------------------

(defun retrieve-primitive-type (exp)
  "Returns the 'primitive' data types."
  (cond ((symbolp exp)
         (get-atomic-type 'symbol))
        ((numberp exp)
         (get-atomic-type 'number))
        ((stringp exp)
         (get-atomic-type 'string))
        (t
         (error "Unidentified Untypable Object: ~s" exp))))

;Function RETRIEVE-PRIMITIVE-TYPE
;<(EXP Sym)> ==>> T-Op
;<(EXP Int)> ==>> T-Op
;<(EXP Str)> ==>> T-Op
```

APPENDIX B

ANALYSIS DECLARATIONS FOR PTI

This appendix section lists the analysis declarations and annotations required for the analysis of PTI by ESSIE.

The first section provides declarations for the global structures, such as `t-op` and `t-var`, used in PTI, as well as the global symbols, such as `*global-env*`.

The second section provides the "normal" annotations required by ESSIE for this system. This consists of pruning some of the strands from `occurs-in-p` that result from the loop-variant type structure, as well as annotating a low-level function (`generic-var-p`) with more accurate information about internal list structure.

The final section provides declarations used to break up chains of mutual recursion, since that facility is not yet completely supported in ESSIE. (Single recursion is fully supported, as the analysis of `prune` and `occurs-in-p` demonstrate.)

```
;;; ----------------------------------------------------------------------
;;; Global data structure definitions
;;; ----------------------------------------------------------------------

(define-defstruct-analysis T-VAR
    :conc-name "T-VAR."
    :constructor (make-t-var (instance))
```

```
    :constraints (or (null instance)
                     (t-var-p instance)
                     (t-op-p instance)))


(define-defstruct-analysis T-OP
    :conc-name "T-OP."
    :constructor (make-t-op (name args))
    :constraints (and (symbolp name)
                      (or (null args)
                          (and (listp args)
                               (or (t-var-p (car args))
                                   (t-op-p (car args)))))))



(define-defstruct-analysis ENV
  :conc-name "ENV."
  :constructor (make-env (identifier t-op))
  :constraints (and (symbolp identifier)
                    (or (t-op-p t-op)
                        (t-var-p t-op))))


(define-symbol-analysis *ATOMIC-TYPE-TABLE*
  :constraints (and (hash-table-p *atomic-type-table*)
                    (t-op-p (gethash 'symbol-key *atomic-type-table*))))


(define-symbol-analysis *GLOBAL-ENV*
  :constraints (and (listp *global-env*)
                    (env-p (car *global-env*))))


(define-symbol-analysis *GENERIC-VAR-LIST*
  :constraints (and (listp *generic-var-list*)
                    (listp (car *generic-var-list*))
                    (t-var-p (car (car *generic-var-list*)))))


;;; ---------------------------------------------------------------------
;;; These two declarations simply make the function annotations easier.
;;; ---------------------------------------------------------------------

(define-symbol-analysis *TYPE-EXP*
  :constraints (or (t-var-p *type-exp*)
                   (t-op-p *type-exp*)))
```

```
(define-symbol-analysis *T-OP*
  :constraints (t-op-p *t-op*))

;;; ---------------------------------------------------------------------
;;; Function annotations
;;; ---------------------------------------------------------------------

;;Prune strands resulting from type-variant recursion from OCCURS-IN-P
(define-fn-analysis OCCURS-IN-P
  #'(lambda (t-var t-exps)
      (cond ((t-var-p t-var)
             (cond ((t-op-p t-exps)
                    *boolean*)
                   ((t-var-p t-exps)
                    *boolean*)
                   ((and (listp t-exps) (t-op-p (car t-exps)))
                    *boolean*)
                   ((and (listp t-exps) (t-var-p (car t-exps)))
                    *boolean*))))))


;;Make GENERIC-VAR-P know that NONGENVARS will be a list of t-vars.
(define-form-in-fn GENERIC-VAR-P nongenvars
  #'(lambda ()
      (list (make-t-var nil))))


;;; ---------------------------------------------------------------------
;;; Function annotations needed to accomodate mutual recursion
;;; (Mutual recursion not completely supported in current implementation.)
;;; ---------------------------------------------------------------------

;;Break multiple recursion between FRESH and FRESH-LIST.
(define-fn-analysis FRESH-LIST
  #'(lambda (type-list nongenvars)
      (t-var.instance (car nongenvars))
      (cond ((null type-list)
             type-list)
            ((t-var-p (car type-list))
             *type-exp*)
            ((t-op-p (car type-list))
             *type-exp*))))

;;break multiple recursion between unify-type and unify-t-op-arg-list
(define-fn-analysis UNIFY-T-OP-ARG-LIST
  #'(lambda (type-exp1 type-exp2)
      (t-op.args type-exp1)
```

```
      (t-op.args type-exp2)
      nil))

;;break multiple recursion between analyze-exp and the other analyzers.
(define-fn-analysis ANALYZE-EXP
  #'(lambda (exp env nongenvars)
      (env.identifier (car env))
      (t-var.instance (car nongenvars))
      (cond ((listp exp)
             *type-exp*)
            ((or (symbolp exp)
                 (numberp exp)
                 (stringp exp))
             *t-op*))))
```

APPENDIX C

EXECUTION STRAND-LEVEL STRUCTURE OF PTI

This section documents the sets of execution strands obtained for the functions in the PTI system by ESSIE.

```
  % -*- Mode:TeX -*-

ANALYZE-FORM
{Fn Br[
  FC: ((LISTP TV-175))
  CE: <(FORM TV-175)>
  RT: T-Var

  FC: ((LISTP TV-197))
  CE: <(FORM TV-197)>
  RT: T-Op-142

  FC: ((STRINGP TV-219) (NOT (NUMBERP TV-219)) (NOT (SYMBOLP TV-219))
       (NOT (LISTP TV-219)))
  CE: <(FORM TV-219)>
  RT: T-Op-153

  FC: ((NUMBERP TV-270) (NOT (SYMBOLP TV-270)) (NOT (LISTP TV-270)))
  CE: <(FORM TV-270)>
  RT: T-Op-163

  FC: ((SYMBOLP TV-312) (NOT (LISTP TV-312)))
  CE: <(FORM TV-312)>
  RT: T-Op-172
  ]}

ANALYZE-DEFUN
{Fn Br[
  FC: ((NOT (T-VAR-P LIST[T-Var])) (NOT (LISTP Sym)) (T-VAR-P T-Var)
       (NOT (T-OP-P T-Var)) (SYMBOLP Sym))
```

```
CE: <(DEFUN-FORM LIST[Sym])(ENV LIST[Env])(NONGENVARS LIST[T-Var])>
RT: T-Var

FC: ((NOT (T-VAR-P LIST[T-Var])) (NOT (LISTP Sym)) (T-VAR-P T-Var)
     (NOT (T-OP-P T-Var)) (SYMBOLP Sym))
CE: <(DEFUN-FORM LIST[Sym])(ENV LIST[Env])(NONGENVARS LIST[T-Var])>
RT: T-Op-9327

FC: ((NOT (T-VAR-P LIST[T-Var])) (NOT (LISTP Sym)) (T-VAR-P T-Var)
     (NOT (T-OP-P T-Var)) (SYMBOLP Sym))
CE: <(DEFUN-FORM LIST[Sym])(ENV LIST[Env])(NONGENVARS LIST[T-Var])>
RT: Boolean[F]

FC: ((NOT (T-VAR-P LIST[T-Var])) (NOT (LISTP Sym)) (T-VAR-P T-Var)
     (NOT (T-OP-P T-Var)) (SYMBOLP Sym))
CE: <(DEFUN-FORM LIST[Sym])(ENV LIST[Env])(NONGENVARS LIST[T-Var])>
RT: Boolean[F]

FC: ((NOT (T-VAR-P LIST[T-Var])) (NOT (LISTP Sym)) (T-VAR-P T-Var)
     (NOT (T-OP-P T-Var)) (SYMBOLP Sym))
CE: <(DEFUN-FORM LIST[Sym])(ENV LIST[Env])(NONGENVARS LIST[T-Var])>
RT: Boolean[F]

FC: ((NOT (T-VAR-P LIST[T-Var])) (NOT (LISTP Sym)) (T-VAR-P T-Var)
     (NOT (T-OP-P T-Var)) (SYMBOLP Sym))
CE: <(DEFUN-FORM LIST[Sym])(ENV LIST[Env])(NONGENVARS LIST[T-Var])>
RT: T-Var

FC: ((NOT (T-VAR-P LIST[T-Var])) (NOT (LISTP Sym)) (T-VAR-P T-Var)
     (NOT (T-OP-P T-Var)) (SYMBOLP Sym))
CE: <(DEFUN-FORM LIST[Sym])(ENV LIST[Env])(NONGENVARS LIST[T-Var])>
RT: T-Op-10578

FC: ((NOT (T-VAR-P LIST[T-Var])) (NOT (LISTP Sym)) (T-VAR-P T-Var)
     (NOT (T-OP-P T-Var)) (SYMBOLP Sym))
CE: <(DEFUN-FORM LIST[Sym])(ENV LIST[Env])(NONGENVARS LIST[T-Var])>
RT: Boolean[F]

FC: ((NOT (T-VAR-P LIST[T-Var])) (NOT (LISTP Sym)) (T-VAR-P T-Var)
     (NOT (T-OP-P T-Var)) (SYMBOLP Sym))
CE: <(DEFUN-FORM LIST[Sym])(ENV LIST[Env])(NONGENVARS LIST[T-Var])>
RT: Boolean[F]

FC: ((NOT (T-VAR-P LIST[T-Var])) (NOT (LISTP Sym)) (T-VAR-P T-Var)
     (NOT (T-OP-P T-Var)) (SYMBOLP Sym))
CE: <(DEFUN-FORM LIST[Sym])(ENV LIST[Env])(NONGENVARS LIST[T-Var])>
RT: Boolean[F]
```

```
FC: ((NOT (T-VAR-P LIST[T-Var])) (NOT (LISTP Sym)) (T-VAR-P T-Var)
     (NOT (T-OP-P T-Var)) (SYMBOLP Sym))
CE: <(DEFUN-FORM LIST[Sym])(ENV LIST[Env])(NONGENVARS LIST[T-Var])>
RT: T-Var

FC: ((NOT (T-VAR-P LIST[T-Var])) (NOT (LISTP Sym)) (T-VAR-P T-Var)
     (NOT (T-OP-P T-Var)) (SYMBOLP Sym))
CE: <(DEFUN-FORM LIST[Sym])(ENV LIST[Env])(NONGENVARS LIST[T-Var])>
RT: T-Op-11829

FC: ((NOT (T-VAR-P LIST[T-Var])) (NOT (LISTP Sym)) (T-VAR-P T-Var)
     (NOT (T-OP-P T-Var)) (SYMBOLP Sym))
CE: <(DEFUN-FORM LIST[Sym])(ENV LIST[Env])(NONGENVARS LIST[T-Var])>
RT: Boolean[F]

FC: ((NOT (T-VAR-P LIST[T-Var])) (NOT (LISTP Sym)) (T-VAR-P T-Var)
     (NOT (T-OP-P T-Var)) (SYMBOLP Sym))
CE: <(DEFUN-FORM LIST[Sym])(ENV LIST[Env])(NONGENVARS LIST[T-Var])>
RT: Boolean[F]

FC: ((NOT (T-VAR-P LIST[T-Var])) (NOT (LISTP Sym)) (T-VAR-P T-Var)
     (NOT (T-OP-P T-Var)) (SYMBOLP Sym))
CE: <(DEFUN-FORM LIST[Sym])(ENV LIST[Env])(NONGENVARS LIST[T-Var])>
RT: Boolean[F]

FC: ((NOT (T-VAR-P LIST[T-Var])) (NOT (LISTP Sym)) (T-VAR-P T-Var)
     (NOT (T-OP-P T-Var)) (SYMBOLP Sym))
CE: <(DEFUN-FORM LIST[Sym])(ENV LIST[Env])(NONGENVARS LIST[T-Var])>
RT: T-Var

FC: ((NOT (T-VAR-P LIST[T-Var])) (NOT (LISTP Sym)) (T-VAR-P T-Var)
     (NOT (T-OP-P T-Var)) (SYMBOLP Sym))
CE: <(DEFUN-FORM LIST[Sym])(ENV LIST[Env])(NONGENVARS LIST[T-Var])>
RT: T-Op-13080

FC: ((NOT (T-VAR-P LIST[T-Var])) (NOT (LISTP Sym)) (T-VAR-P T-Var)
     (NOT (T-OP-P T-Var)) (SYMBOLP Sym))
CE: <(DEFUN-FORM LIST[Sym])(ENV LIST[Env])(NONGENVARS LIST[T-Var])>
RT: Boolean[F]

FC: ((NOT (T-VAR-P LIST[T-Var])) (NOT (LISTP Sym)) (T-VAR-P T-Var)
     (NOT (T-OP-P T-Var)) (SYMBOLP Sym))
CE: <(DEFUN-FORM LIST[Sym])(ENV LIST[Env])(NONGENVARS LIST[T-Var])>
RT: Boolean[F]

FC: ((NOT (T-VAR-P LIST[T-Var])) (NOT (LISTP Sym)) (T-VAR-P T-Var)
     (NOT (T-OP-P T-Var)) (SYMBOLP Sym))
CE: <(DEFUN-FORM LIST[Sym])(ENV LIST[Env])(NONGENVARS LIST[T-Var])>
```

```
RT: Boolean[F]

FC: ((NOT (T-VAR-P LIST[T-Var])) (NOT (LISTP Sym)) (T-VAR-P T-Var)
     (NOT (T-OP-P T-Var)) (SYMBOLP Sym))
CE: <(DEFUN-FORM LIST[Sym])(ENV LIST[Env])(NONGENVARS LIST[T-Var])>
RT: T-Var

FC: ((NOT (T-VAR-P LIST[T-Var])) (NOT (LISTP Sym)) (T-VAR-P T-Var)
     (NOT (T-OP-P T-Var)) (SYMBOLP Sym))
CE: <(DEFUN-FORM LIST[Sym])(ENV LIST[Env])(NONGENVARS LIST[T-Var])>
RT: T-Op-14331

FC: ((NOT (T-VAR-P LIST[T-Var])) (NOT (LISTP Sym)) (T-VAR-P T-Var)
     (NOT (T-OP-P T-Var)) (SYMBOLP Sym))
CE: <(DEFUN-FORM LIST[Sym])(ENV LIST[Env])(NONGENVARS LIST[T-Var])>
RT: Boolean[F]

FC: ((NOT (T-VAR-P LIST[T-Var])) (NOT (LISTP Sym)) (T-VAR-P T-Var)
     (NOT (T-OP-P T-Var)) (SYMBOLP Sym))
CE: <(DEFUN-FORM LIST[Sym])(ENV LIST[Env])(NONGENVARS LIST[T-Var])>
RT: Boolean[F]

FC: ((NOT (T-VAR-P LIST[T-Var])) (NOT (LISTP Sym)) (T-VAR-P T-Var)
     (NOT (T-OP-P T-Var)) (SYMBOLP Sym))
CE: <(DEFUN-FORM LIST[Sym])(ENV LIST[Env])(NONGENVARS LIST[T-Var])>
RT: Boolean[F]

FC: ((NOT (T-VAR-P LIST[T-Var])) (NOT (LISTP Sym)) (T-VAR-P T-Var)
     (NOT (T-OP-P T-Var)) (SYMBOLP Sym))
CE: <(DEFUN-FORM LIST[Sym])(ENV LIST[Env])(NONGENVARS LIST[T-Var])>
RT: T-Var

FC: ((NOT (T-VAR-P LIST[T-Var])) (NOT (LISTP Sym)) (T-VAR-P T-Var)
     (NOT (T-OP-P T-Var)) (SYMBOLP Sym))
CE: <(DEFUN-FORM LIST[Sym])(ENV LIST[Env])(NONGENVARS LIST[T-Var])>
RT: T-Op-15582

FC: ((NOT (T-VAR-P LIST[T-Var])) (NOT (LISTP Sym)) (T-VAR-P T-Var)
     (NOT (T-OP-P T-Var)) (SYMBOLP Sym))
CE: <(DEFUN-FORM LIST[Sym])(ENV LIST[Env])(NONGENVARS LIST[T-Var])>
RT: Boolean[F]

FC: ((NOT (T-VAR-P LIST[T-Var])) (NOT (LISTP Sym)) (T-VAR-P T-Var)
     (NOT (T-OP-P T-Var)) (SYMBOLP Sym))
CE: <(DEFUN-FORM LIST[Sym])(ENV LIST[Env])(NONGENVARS LIST[T-Var])>
RT: Boolean[F]

FC: ((NOT (T-VAR-P LIST[T-Var])) (NOT (LISTP Sym)) (T-VAR-P T-Var)
```

```
      (NOT (T-OP-P T-Var)) (SYMBOLP Sym))
  CE: <(DEFUN-FORM LIST[Sym])(ENV LIST[Env])(NONGENVARS LIST[T-Var])>
  RT: Boolean[F]
  ]}

ANALYZE-LET
{Fn Br[
  FC: ((LISTP LIST[LIST[TV-3579]]) (SYMBOLP TV-3579) (NOT (LISTP TV-3579)))
  CE: <(LET-FORM LIST[LIST[LIST[TV-3579]]])(ENV LIST[Env])
      (NONGENVARS LIST[T-Var])>
  RT: T-Op-3381

  FC: ((LISTP LIST[LIST[TV-3635]]) (SYMBOLP TV-3635) (NOT (LISTP TV-3635)))
  CE: <(LET-FORM LIST[LIST[LIST[TV-3635]]])(ENV LIST[Env])
      (NONGENVARS LIST[T-Var])>
  RT: T-Var

  FC: ((LISTP LIST[LIST[TV-4272]]) (NUMBERP TV-4272) (NOT (SYMBOLP TV-4272))
      (NOT (LISTP TV-4272)))
  CE: <(LET-FORM LIST[LIST[LIST[TV-4272]]])(ENV LIST[Env])
      (NONGENVARS LIST[T-Var])>
  RT: T-Op-4063

  FC: ((LISTP LIST[LIST[TV-4333]]) (NUMBERP TV-4333) (NOT (SYMBOLP TV-4333))
      (NOT (LISTP TV-4333)))
  CE: <(LET-FORM LIST[LIST[LIST[TV-4333]]])(ENV LIST[Env])
      (NONGENVARS LIST[T-Var])>
  RT: T-Var

  FC: ((LISTP LIST[LIST[TV-5071]]) (STRINGP TV-5071) (NOT (NUMBERP TV-5071))
      (NOT (SYMBOLP TV-5071)) (NOT (LISTP TV-5071)))
  CE: <(LET-FORM LIST[LIST[LIST[TV-5071]]])(ENV LIST[Env])
      (NONGENVARS LIST[T-Var])>
  RT: T-Op-4853

  FC: ((LISTP LIST[LIST[TV-5137]]) (STRINGP TV-5137) (NOT (NUMBERP TV-5137))
      (NOT (SYMBOLP TV-5137)) (NOT (LISTP TV-5137)))
  CE: <(LET-FORM LIST[LIST[LIST[TV-5137]]])(ENV LIST[Env])
      (NONGENVARS LIST[T-Var])>
  RT: T-Var

  FC: ((LISTP LIST[LIST[TV-5798]]) (LISTP TV-5798))
  CE: <(LET-FORM LIST[LIST[LIST[TV-5798]]])(ENV LIST[Env])
      (NONGENVARS LIST[T-Var])>
  RT: T-Op-5621

  FC: ((LISTP LIST[LIST[TV-5852]]) (LISTP TV-5852))
  CE: <(LET-FORM LIST[LIST[LIST[TV-5852]]])(ENV LIST[Env])
```

```
      (NONGENVARS LIST[T-Var])>
  RT: T-Var
  ]}

ANALYZE-FUNCALL
{Fn Br[
  FC: ((LISTP TV-85))
  CE: <(FUNCALL-FORM LIST[TV-85])(ENV LIST[Env])(NONGENVARS LIST[T-Var])>
  RT: T-Op-53

  FC: ((LISTP TV-107))
  CE: <(FUNCALL-FORM LIST[TV-107])(ENV LIST[Env])(NONGENVARS LIST[T-Var])>
  RT: T-Var

  FC: ((SYMBOLP TV-129) (NOT (LISTP TV-129)))
  CE: <(FUNCALL-FORM LIST[TV-129])(ENV LIST[Env])(NONGENVARS LIST[T-Var])>
  RT: T-Op-72

  FC: ((SYMBOLP TV-162) (NOT (LISTP TV-162)))
  CE: <(FUNCALL-FORM LIST[TV-162])(ENV LIST[Env])(NONGENVARS LIST[T-Var])>
  RT: T-Var
  ]}

ANALYZE-APPLICATION
{Fn Br[
  FC: ((LISTP TV-6429))
  CE: <(APPL LIST[TV-6429])(ENV LIST[Env])(NONGENVARS LIST[T-Var])>
  RT: T-Op-6347

  FC: ((LISTP TV-6479))
  CE: <(APPL LIST[TV-6479])(ENV LIST[Env])(NONGENVARS LIST[T-Var])>
  RT: T-Var

  FC: ((SYMBOLP TV-15199) (NOT (LISTP TV-15199)))
  CE: <(APPL LIST[TV-15199])(ENV LIST[Env])(NONGENVARS LIST[T-Var])>
  RT: T-Op-15112

  FC: ((SYMBOLP TV-15253) (NOT (LISTP TV-15253)))
  CE: <(APPL LIST[TV-15253])(ENV LIST[Env])(NONGENVARS LIST[T-Var])>
  RT: T-Var
  ]}

ANALYZE-PROGN
{Fn Br[
  FC: ((NOT (NULL LIST[TV-5540])) (SYMBOLP TV-5540) (NOT (LISTP TV-5540)))
  CE: <(PROGN-FORM LIST[TV-5540])(ENV LIST[Env])(NONGENVARS LIST[T-Var])>
  RT: T-Op-5561
```

```
  FC: ((NOT (NULL LIST[TV-5102])) (NUMBERP TV-5102) (NOT (SYMBOLP TV-5102))
       (NOT (LISTP TV-5102)))
  CE: <(PROGN-FORM LIST[TV-5102])(ENV LIST[Env])(NONGENVARS LIST[T-Var])>
  RT: T-Op-5124

  FC: ((NOT (NULL LIST[TV-4637])) (STRINGP TV-4637) (NOT (NUMBERP TV-4637))
       (NOT (SYMBOLP TV-4637)) (NOT (LISTP TV-4637)))
  CE: <(PROGN-FORM LIST[TV-4637])(ENV LIST[Env])(NONGENVARS LIST[T-Var])>
  RT: T-Op-4660

  FC: ((NOT (NULL LIST[TV-4223])) (LISTP TV-4223))
  CE: <(PROGN-FORM LIST[TV-4223])(ENV LIST[Env])(NONGENVARS LIST[T-Var])>
  RT: T-Op-4243

  FC: ((NOT (NULL LIST[TV-4159])) (LISTP TV-4159))
  CE: <(PROGN-FORM LIST[TV-4159])(ENV LIST[Env])(NONGENVARS LIST[T-Var])>
  RT: T-Var
  ]}

ANALYZE-IF
{Fn Br[
  FC: ((SYMBOLP TV-13784) (NOT (LISTP TV-13784)))
  CE: <(IF-FORM LIST[TV-13784])(ENV LIST[Env])(NONGENVARS LIST[T-Var])>
  RT: T-Op-13790

  FC: ((NUMBERP TV-14850) (NOT (SYMBOLP TV-14850)) (NOT (LISTP TV-14850)))
  CE: <(IF-FORM LIST[TV-14850])(ENV LIST[Env])(NONGENVARS LIST[T-Var])>
  RT: T-Op-14856

  FC: ((STRINGP TV-16048) (NOT (NUMBERP TV-16048)) (NOT (SYMBOLP TV-16048))
       (NOT (LISTP TV-16048)))
  CE: <(IF-FORM LIST[TV-16048])(ENV LIST[Env])(NONGENVARS LIST[T-Var])>
  RT: T-Op-16054

  FC: ((LISTP TV-21441))
  CE: <(IF-FORM LIST[TV-21441])(ENV LIST[Env])(NONGENVARS LIST[T-Var])>
  RT: T-Op-21447

  FC: ((LISTP TV-22048))
  CE: <(IF-FORM LIST[TV-22048])(ENV LIST[Env])(NONGENVARS LIST[T-Var])>
  RT: T-Var
  ]}

ANALYZE-EXP
{Fn Br[
  FC: ((LISTP TV-2121))
  CE: <(EXP TV-2121)(ENV LIST[Env])(NONGENVARS LIST[T-Var])>
  RT: T-Var
```

```
  FC: ((LISTP TV-2128))
  CE: <(EXP TV-2128)(ENV LIST[Env])(NONGENVARS LIST[T-Var])>
  RT: T-Op-2116

  FC: ((STRINGP TV-2342) (NOT (NUMBERP TV-2342)) (NOT (SYMBOLP TV-2342))
       (NOT (LISTP TV-2342)))
  CE: <(EXP TV-2342)(ENV LIST[Env])(NONGENVARS LIST[T-Var])>
  RT: T-Op-2334

  FC: ((NUMBERP TV-2560) (NOT (SYMBOLP TV-2560)) (NOT (LISTP TV-2560)))
  CE: <(EXP TV-2560)(ENV LIST[Env])(NONGENVARS LIST[T-Var])>
  RT: T-Op-2553

  FC: ((SYMBOLP TV-2676) (NOT (LISTP TV-2676)))
  CE: <(EXP TV-2676)(ENV LIST[Env])(NONGENVARS LIST[T-Var])>
  RT: T-Op-2670
  ]}


(FUNCALL-P
{Fn B[
  FC: ((LISTP LIST[Sym]))
  CE: <(EXP LIST[Sym])>
  RT: Boolean[T]

  FC: ((NOT (LISTP TV-326)))
  CE: <(EXP TV-326)>
  RT: Boolean[F]

  FC: ((LISTP LIST[TV-445]))
  CE: <(EXP LIST[TV-445])>
  RT: Boolean[F]

  FC: ((LISTP LIST[Sym]))
  CE: <(EXP LIST[Sym])>
  RT: Boolean[F]
  ]}


LET-P
{Fn Br[
  FC: ((LISTP LIST[Sym]))
  CE: <(EXP LIST[Sym])>
  RT: Boolean[T]

  FC: ((NOT (LISTP TV-326)))
  CE: <(EXP TV-326)>
```

```
  RT: Boolean[F]

  FC: ((LISTP LIST[TV-445]))
  CE: <(EXP LIST[TV-445])>
  RT: Boolean[F]

  FC: ((LISTP LIST[Sym]))
  CE: <(EXP LIST[Sym])>
  RT: Boolean[F]
  ]}

DEFUN-P
{Fn Br[
  FC: ((LISTP LIST[Sym]))
  CE: <(EXP LIST[Sym])>
  RT: Boolean[T]

  FC: ((NOT (LISTP TV-326)))
  CE: <(EXP TV-326)>
  RT: Boolean[F]

  FC: ((LISTP LIST[TV-445]))
  CE: <(EXP LIST[TV-445])>
  RT: Boolean[F]

  FC: ((LISTP LIST[Sym]))
  CE: <(EXP LIST[Sym])>
  RT: Boolean[F]
  ]}

IF-P
{Fn Br[
  FC: ((LISTP LIST[Sym]))
  CE: <(EXP LIST[Sym])>
  RT: Boolean[T]

  FC: ((NOT (LISTP TV-326)))
  CE: <(EXP TV-326)>
  RT: Boolean[F]

  FC: ((LISTP LIST[TV-445]))
  CE: <(EXP LIST[TV-445])>
  RT: Boolean[F]

  FC: ((LISTP LIST[Sym]))
  CE: <(EXP LIST[Sym])>
  RT: Boolean[F]
  ]}
```

```
UNIFY-TYPE
{Fn Br[
  FC: ((NOT (T-VAR-P T-Op-33196)) (T-VAR-P T-Var) (NOT (T-OP-P T-Var))
       (T-OP-P T-Op-33196))
  CE: <(TYPE-EXP1 T-Var)(TYPE-EXP2 T-Op-33196)>
  RT: Nil

  FC: ((T-VAR-P T-Var) (NOT (T-OP-P T-Var)))
  CE: <(TYPE-EXP1 T-Var)(TYPE-EXP2 T-Var)>
  RT: Nil

  FC: ((T-VAR-P T-Var) (NOT (T-OP-P T-Var)) (T-OP-P TV-49478))
  CE: <(TYPE-EXP1 T-Var)(TYPE-EXP2 TV-49478)>
  RT: Nil

  FC: ((NOT (T-VAR-P T-Op-57133)) (NOT (T-VAR-P T-Op-57132))
       (T-OP-P T-Op-57132) (T-OP-P T-Op-57133))
  CE: <(TYPE-EXP1 T-Op-57132)(TYPE-EXP2 T-Op-57133)>
  RT: Nil

  FC: ((NOT (T-VAR-P TV-59414)) (T-VAR-P T-Var) (NOT (T-OP-P T-Var))
       (T-OP-P TV-59414))
  CE: <(TYPE-EXP1 TV-59414)(TYPE-EXP2 T-Var)>
  RT: Nil

  FC: ((NOT (T-VAR-P T-Op-59275)) (T-VAR-P T-Var) (NOT (T-OP-P T-Var))
       (T-OP-P T-Op-59275))
  CE: <(TYPE-EXP1 T-Op-59275)(TYPE-EXP2 T-Var)>
  RT: Nil

  FC: ((T-VAR-P T-Var) (NOT (T-OP-P T-Var)) (T-VAR-P T-Var)
       (NOT (T-OP-P T-Var)))
  CE: <(TYPE-EXP1 T-Var)(TYPE-EXP2 T-Var)>
  RT: Nil
  ]}

UNIFY-T-OP-ARG-LIST
{Fn Br[
  FC: NIL
  CE: <(TYPE-EXP1 T-Op-283)(TYPE-EXP2 T-Op-284)>
  RT: Nil
  ]}


LIST
{Fn Br[
  FC: NIL
```

```
  CE: <(SE::X TV-11)(SE::Y TV-11)>
  RT: LIST[TV-11]
  ]}

CAR
{Fn Br[
  FC: NIL
  CE: <(SE::X LIST[TV-5])>
  RT: TV-5
  ]}

LENGTH
{Fn Br[
  FC: ((NULL TV-42))
  CE: <(SE::LST TV-42)>
  RT: Int

  FC: ((NOT (NULL LIST[TV-63])))
  CE: <(SE::LST LIST[TV-63])>
  RT: Int
  ]}

RETRIEVE-TYPE
{Fn Br[
  FC: NIL
  CE: <(IDENTIFIER Sym)(ENV LIST[Env])(NONGENVARS LIST[T-Var])>
  RT: T-Var

  FC: NIL
  CE: <(IDENTIFIER Sym)(ENV LIST[Env])(NONGENVARS LIST[T-Var])>
  RT: T-Op-1714

  FC: NIL
  CE: <(IDENTIFIER Sym)(ENV LIST[Env])(NONGENVARS LIST[T-Var])>
  RT: Boolean[F]

  FC: NIL
  CE: <(IDENTIFIER TV-2034)(ENV LIST[Env])(NONGENVARS TV-2037)>
  RT: Boolean[F]

  FC: NIL
  CE: <(IDENTIFIER Sym)(ENV LIST[Env])(NONGENVARS TV-2069)>
  RT: Boolean[F]
  ]}

FRESH-TYPE
{Fn Br[
  FC: ((NOT (T-VAR-P T-Op-50)) (T-OP-P T-Op-50))
```

```
  CE: <(TYPE-EXP T-Op-50)(NONGENVARS LIST[T-Var])>
  RT: T-Op-53

  FC: ((T-VAR-P T-Var) (NOT (T-OP-P T-Var)))
  CE: <(TYPE-EXP T-Var)(NONGENVARS LIST[T-Var])>
  RT: T-Var

  FC: ((T-VAR-P T-Var) (NOT (T-OP-P T-Var)))
  CE: <(TYPE-EXP T-Var)(NONGENVARS LIST[T-Var])>
  RT: T-Op-64

  FC: ((T-VAR-P T-Var) (NOT (T-OP-P T-Var)))
  CE: <(TYPE-EXP T-Var)(NONGENVARS LIST[T-Var])>
  RT: T-Var
  ]}

FRESH
{Fn Br[
  FC: ((NOT (T-VAR-P T-Op-1653)) (T-OP-P T-Op-1653))
  CE: <(TYPE-EXP T-Op-1653)(NONGENVARS LIST[T-Var])>
  RT: T-Op-1590

  FC: ((T-VAR-P T-Var) (NOT (T-OP-P T-Var)))
  CE: <(TYPE-EXP T-Var)(NONGENVARS LIST[T-Var])>
  RT: T-Var

  FC: ((T-VAR-P T-Var) (NOT (T-OP-P T-Var)))
  CE: <(TYPE-EXP T-Var)(NONGENVARS LIST[T-Var])>
  RT: T-Op-4425

  FC: ((T-VAR-P T-Var) (NOT (T-OP-P T-Var)))
  CE: <(TYPE-EXP T-Var)(NONGENVARS LIST[T-Var])>
  RT: T-Var
  ]}

FRESH-LIST
{Fn Br[
  FC: ((NULL TV-1152))
  CE: <(TYPE-LIST TV-1152)(NONGENVARS LIST[T-Var])>
  RT: TV-1152

  FC: ((T-VAR-P TV-1287) (NOT (NULL LIST[TV-1287])))
  CE: <(TYPE-LIST LIST[TV-1287])(NONGENVARS LIST[T-Var])>
  RT: T-Var

  FC: ((T-VAR-P TV-1293) (NOT (NULL LIST[TV-1293])))
  CE: <(TYPE-LIST LIST[TV-1293])(NONGENVARS LIST[T-Var])>
  RT: T-Op-1282
```

```
  FC: ((T-OP-P TV-1505) (NOT (T-VAR-P TV-1505)) (NOT (NULL LIST[TV-1505])))
  CE: <(TYPE-LIST LIST[TV-1505])(NONGENVARS LIST[T-Var])>
  RT: T-Var

  FC: ((T-OP-P TV-1512) (NOT (T-VAR-P TV-1512)) (NOT (NULL LIST[TV-1512])))
  CE: <(TYPE-LIST LIST[TV-1512])(NONGENVARS LIST[T-Var])>
  RT: T-Op-1499
  ]}

GENERIC-VAR-P
{Fn Br[
  FC: NIL
  CE: <(T-VAR T-Var)(NONGENVARS LIST[T-Var])>
  RT: Boolean[T]

  FC: NIL
  CE: <(T-VAR T-Var)(NONGENVARS LIST[T-Var])>
  RT: Boolean[F]
  ]}

OCCURS-IN-P
{Fn Br[
  FC: ((T-OP-P TV-253) (T-VAR-P TV-252))
  CE: <(T-VAR TV-252)(T-EXPS TV-253)>
  RT: Boolean[T]

  FC: ((T-OP-P TV-257) (T-VAR-P TV-256))
  CE: <(T-VAR TV-256)(T-EXPS TV-257)>
  RT: Boolean[F]

  FC: ((T-VAR-P TV-403) (NOT (T-OP-P TV-403)) (T-VAR-P TV-402))
  CE: <(T-VAR TV-402)(T-EXPS TV-403)>
  RT: Boolean[T]

  FC: ((T-VAR-P TV-408) (NOT (T-OP-P TV-408)) (T-VAR-P TV-407))
  CE: <(T-VAR TV-407)(T-EXPS TV-408)>
  RT: Boolean[F]

  FC: ((T-OP-P TV-785) (LISTP LIST[TV-785]) (NOT (T-VAR-P LIST[TV-785]))
       (NOT (T-OP-P LIST[TV-785])) (T-VAR-P TV-784))
  CE: <(T-VAR TV-784)(T-EXPS LIST[TV-785])>
  RT: Boolean[T]

  FC: ((T-OP-P TV-794) (LISTP LIST[TV-794]) (NOT (T-VAR-P LIST[TV-794]))
       (NOT (T-OP-P LIST[TV-794])) (T-VAR-P TV-793))
  CE: <(T-VAR TV-793)(T-EXPS LIST[TV-794])>
  RT: Boolean[F]
```

```
  FC: ((T-VAR-P TV-1705) (NOT (T-OP-P TV-1705)) (LISTP LIST[TV-1705])
       (NOT (T-VAR-P LIST[TV-1705])) (NOT (T-OP-P LIST[TV-1705]))
       (T-VAR-P TV-1704))
  CE: <(T-VAR TV-1704)(T-EXPS LIST[TV-1705])>
  RT: Boolean[T]

  FC: ((T-VAR-P TV-1715) (NOT (T-OP-P TV-1715)) (LISTP LIST[TV-1715])
       (NOT (T-VAR-P LIST[TV-1715])) (NOT (T-OP-P LIST[TV-1715]))
       (T-VAR-P TV-1714))
  CE: <(T-VAR TV-1714)(T-EXPS LIST[TV-1715])>
  RT: Boolean[F]
  ]}


RETRIEVE-PRIMITIVE-TYPE
{Fn Br[
  FC: ((SYMBOLP TV-104))
  CE: <(EXP TV-104)>
  RT: T-Op-100

  FC: ((NUMBERP TV-159) (NOT (SYMBOLP TV-159)))
  CE: <(EXP TV-159)>
  RT: T-Op-154

  FC: ((STRINGP TV-276) (NOT (NUMBERP TV-276)) (NOT (SYMBOLP TV-276)))
  CE: <(EXP TV-276)>
  RT: T-Op-270
  ]}

FRESH-VAR
{Fn Br[
  FC: NIL
  CE: <(T-VAR T-Var)>
  RT: T-Var
  ]}

EXTEND-ENV
{Fn Br[
  FC: ((T-OP-P TV-626) (SYMBOLP TV-625) (NOT (LISTP TV-626))
       (NOT (LISTP TV-625)))
  CE: <(IDENTIFIERS TV-625)(TYPES TV-626)(ENV LIST[Env])>
  RT: LIST[Env]

  FC: ((T-VAR-P TV-646) (NOT (T-OP-P TV-646)) (SYMBOLP TV-645)
       (NOT (LISTP TV-646)) (NOT (LISTP TV-645)))
  CE: <(IDENTIFIERS TV-645)(TYPES TV-646)(ENV LIST[Env])>
  RT: LIST[Env]
```

```
  FC: ((T-OP-P TV-1650) (SYMBOLP TV-1648) (LISTP LIST[TV-1650])
       (LISTP LIST[TV-1648]))
  CE: <(IDENTIFIERS LIST[TV-1648])(TYPES LIST[TV-1650])(ENV LIST[Env])>
  RT: LIST[Env]

  FC: ((T-VAR-P TV-1673) (NOT (T-OP-P TV-1673)) (SYMBOLP TV-1671)
       (LISTP LIST[TV-1673]) (LISTP LIST[TV-1671]))
  CE: <(IDENTIFIERS LIST[TV-1671])(TYPES LIST[TV-1673])(ENV LIST[Env])>
  RT: LIST[Env]
  ]}

GET-GLOBAL-BINDING
{Fn Br[
  FC: NIL
  CE: <(IDENTIFIER Sym)(ENV LIST[Env])>
  RT: T-Var

  FC: NIL
  CE: <(IDENTIFIER Sym)(ENV LIST[Env])>
  RT: T-Op-395

  FC: NIL
  CE: <(IDENTIFIER TV-428)(ENV LIST[Env])>
  RT: Nil

  FC: NIL
  CE: <(IDENTIFIER Sym)(ENV LIST[Env])>
  RT: Nil
  ]}

EXTEND-NONGENVARS
{Fn Br[
  FC: ((T-VAR-P TV-69))
  CE: <(VARS TV-69)(NONGENVARS LIST[TV-69])>
  RT: LIST[TV-69]

  FC: ((NOT (T-VAR-P LIST[TV-139])))
  CE: <(VARS LIST[TV-139])(NONGENVARS LIST[TV-139])>
  RT: LIST[TV-139]
  ]}

SETUP
{Fn Br[
  FC: NIL
  CE: <Empty>
  RT: LIST[Env]
  ]}
```

```
ADD-GLOBAL-BINDING
{Fn Br[
  FC: ((SYMBOLP TV-1290) (T-OP-P TV-1286) (NOT (T-VAR-P TV-1286))
       (LISTP LIST[TV-1286]) (NOT (NULL LIST[TV-1286])) (SYMBOLP TV-1291))
  CE: <(IDENTIFIER TV-1290)(TYPE-NAME TV-1291)(ARG-LIST LIST[TV-1286])>
  RT: LIST[Env]

  FC: ((SYMBOLP TV-1308) (T-VAR-P TV-1304) (LISTP LIST[TV-1304])
       (NOT (NULL LIST[TV-1304])) (SYMBOLP TV-1309))
  CE: <(IDENTIFIER TV-1308)(TYPE-NAME TV-1309)(ARG-LIST LIST[TV-1304])>
  RT: LIST[Env]

  FC: ((SYMBOLP TV-1322) (NULL TV-1324) (SYMBOLP TV-1323))
  CE: <(IDENTIFIER TV-1322)(TYPE-NAME TV-1323)(ARG-LIST TV-1324)>
  RT: LIST[Env]
  ]}

MAKE-ENV
{Fn Br[
  FC: ((T-VAR-P TV-654) (NOT (T-OP-P TV-654)) (SYMBOLP TV-655))
  CE: <(T-OP TV-654)(IDENTIFIER TV-655)>
  RT: Env

  FC: ((T-OP-P TV-660) (SYMBOLP TV-661))
  CE: <(T-OP TV-660)(IDENTIFIER TV-661)>
  RT: Env
  ]}

ENV.T-OP
{Fn MAKE-ENV.T-OP-BRAID}

(ENV.IDENTIFIER
{Fn MAKE-ENV.IDENTIFIER-BRAID}

*GLOBAL-ENV*
{Fn *GLOBAL-ENV*-SYMBOL-BRAID-CONSTRUCTOR}

MAKE-T-OP
{Fn Br[
  FC: ((T-OP-P TV-847) (NOT (T-VAR-P TV-847)) (LISTP LIST[TV-847])
       (NOT (NULL LIST[TV-847])) (SYMBOLP TV-849))
  CE: <(ARGS LIST[TV-847])(NAME TV-849)>
  RT: T-Op-840

  FC: ((T-VAR-P TV-857) (LISTP LIST[TV-857]) (NOT (NULL LIST[TV-857]))
       (SYMBOLP TV-859))
  CE: <(ARGS LIST[TV-857])(NAME TV-859)>
```

```
  RT: T-Op-851

  FC: ((NULL TV-864) (SYMBOLP TV-865))
  CE: <(ARGS TV-864)(NAME TV-865)>
  RT: T-Op-861
  ]}

T-OP.ARGS
{Fn MAKE-T-OP.ARGS-BRAID}

T-OP.NAME
{Fn MAKE-T-OP.NAME-BRAID}

SETUP-ATOMIC-TYPES
{Fn Br[
  FC: NIL
  CE: <Empty>
  RT: Sym
  ]}

ADD-ATOMIC-TYPE
{Fn Br[
  FC: NIL
  CE: <(IDENTIFIER Sym)>
  RT: T-Op-357
  ]}

GET-ATOMIC-TYPE
{Fn Br[
  FC: NIL
  CE: <(NAME Sym)>
  RT: T-Op-202
  ]}

PRUNE
{Fn Br[
  FC: ((T-OP-P TV-93))
  CE: <(TYPE-EXP TV-93)>
  RT: TV-93

  FC: ((T-VAR-P T-Var) (NOT (T-OP-P T-Var)))
  CE: <(TYPE-EXP T-Var)>
  RT: T-Var

  FC: ((T-VAR-P T-Var) (NOT (T-OP-P T-Var)))
  CE: <(TYPE-EXP T-Var)>
  RT: T-Op-1228
```

```
  FC: ((T-VAR-P T-Var) (NOT (T-OP-P T-Var)))
  CE: <(TYPE-EXP T-Var)>
  RT: T-Var
  ]}

MAKE-T-VAR
{Fn Br[
  FC: ((NULL TV-340))
  CE: <(INSTANCE TV-340)>
  RT: T-Var

  FC: ((T-VAR-P TV-345) (NOT (NULL TV-345)))
  CE: <(INSTANCE TV-345)>
  RT: T-Var

  FC: ((T-OP-P TV-351) (NOT (T-VAR-P TV-351)) (NOT (NULL TV-351)))
  CE: <(INSTANCE TV-351)>
  RT: T-Var
  ]}

*GENERIC-VAR-LIST*
{Fn *GENERIC-VAR-LIST*-SYMBOL-BRAID-CONSTRUCTOR}

*ATOMIC-TYPE-TABLE*
{Fn *ATOMIC-TYPE-TABLE*-SYMBOL-BRAID-CONSTRUCTOR}

T-VAR.INSTANCE
{Fn MAKE-T-VAR.INSTANCE-BRAID}
```

APPENDIX D

REPRESENTATIVE ANALYZERS AND ANALYSIS OBJECTS

This section provides examples of some of the analyzers and analysis objects in the current type analysis system implemented with ESSIE.

```
;;; ---------------------------------------------------------------------
;;; The analysis definition for type variables
;;; ---------------------------------------------------------------------

(define-analysis type-var

  :unifier
  #'(lambda (analysis1 analysis2)
      "Unify if either arg is a type var.
       Returns a subst structure."
      (cond ((not (or (type-var-p analysis1) (type-var-p analysis2)))
             (make-subst :success nil))
            ((eq analysis1 analysis2)
             (make-subst :success t :list nil))
            ((or (c-type-var-p analysis1) (c-type-var-p analysis2))
             (make-subst :success nil :list nil))
            ((type-var-p analysis1)
             (if (occurs-in-analysis-p analysis1 analysis2)
                 (essie-error "Unification occurs check error.")
                 (make-subst :success T :list  '((,analysis1 ,analysis2)))))
            ((type-var-p analysis2)
             (if (occurs-in-analysis-p analysis2 analysis1)
                 (essie-error "Unification occurs check error.")
                 (make-subst :success T :list '((,analysis2 ,analysis1)))))
            ))

  :equal-p
  #'(lambda (type-var1 type-var2)
      "Two type vars are always structurally equal"
      (declare (ignore type-var1 type-var2))
```

```
      T)

  :copier
  #'(lambda (type-var env)
      "Returns a freshened copy of TYPE-VAR, with all t-vars not appearing
       in ENV replaced."
      (if (generic-var-p type-var env)
          (fresh-var type-var)
          type-var))


  :component-p
  #'(lambda (tvar analysis)
      "Returns T if analysis is a component of TVAR."
      (eq tvar analysis))


  :print-function print-type-var
  )

;;; ---------------------------------------------------------------------
;;; The analysis definition for "type operators"---simple data objects
;;;     such as those represented in parametric polymorphic type inference.
;;; ---------------------------------------------------------------------

(define-analysis type-op

  :slots ((name symbolp)
          (args (list-of analysis-p)))

  :unifier
  #'(lambda (analysis1 analysis2)
      "Unifies 2 type-op analysis structures.
       Returns a subst structure."
      (cond ((and (type-op-p analysis1) (type-op-p analysis2)
                  (equal (type-op.name analysis1) (type-op.name analysis2)))
             (if (null (type-op.args analysis1))
                 (make-subst :success t :list nil)
                 (unify-analysis-lists (type-op.args analysis1)
                                       (type-op.args analysis2))))
            (t ;;different type ops
             (make-subst :success nil))))

  :equal-p
  #'(lambda (analysis1 analysis2)
      (and (equal (type-op.name analysis1) (type-op.name analysis2))
           (uel:mapc-and #'(lambda (element1 element2)
                             (equal-analyses-p element1 element2))
```

```
                        (type-op.args analysis1) (type-op.args analysis2))))

  :copier
  #'(lambda (t-op env)
      "Returns a freshened copy of T-OP, copying all t-vars not in ENV."
      (make-type-op
        :name (type-op.name t-op)
        :args (fresh-list (type-op.args t-op) env)))

  :occurs-checker
  #'(lambda (t-op type-var)
      "Returns T if TYPE-VAR occurs in T-OP, NIL otherwise."
      (uel:mapc-or #'(lambda (analysis)
                       (occurs-in-analysis-p type-var analysis))
                   (type-op.args t-op)))

  :component-p
   #'(lambda (type-op analysis)
       (or (eq (prune type-op) (prune analysis))
           (some #'(lambda (type-op-component)
                     (send-analysis :component-p (prune type-op-component)
                                                 (prune analysis)))
                 (type-op.args type-op))))

  :print-function print-type-op
  )


;;; ---------------------------------------------------------------------
;;; An example definition of a data object lacking internal structure.
;;; ---------------------------------------------------------------------

(define-primitive-analysis string-analysis
  :print-string "Str")

;;; ---------------------------------------------------------------------
;;; The analyzer definition for LET.
;;; ---------------------------------------------------------------------

(define-analyzer let (let-form braid/strand)
  "Returns the braid resulting from analysis of LET-FORM in BRAID/STRAND."

  (with-strand (strand braid/strand)
    (let* ((bindings (second let-form))
           (idents (mapcar #'first-if-list bindings))
           (let-body (cddr let-form))
           (binding-braids
             (mapcar
```

```
                #'(lambda (binding)
                    (if (and (listp binding) (second binding))
                        (analyze-exp (second binding)
                                     (fresh-strand strand))
                        (analysis->result (make-type-var)
                                          (fresh-strand strand))))
                  bindings)))

      (with-arg-analyses ((fc env args) binding-braids)
        ;;create a strand with the augmented environment.
        (let ((new-strand (make-strand
                           :fc fc
                           :env env
                           :result (make-type-var))))
          (push-env env idents args)
          (let ((final-braid (call-analyzer 'progn let-body new-strand)))
            ;;pop off the let bindings
            (mapc #'(lambda (strand)
                      (pop-env (strand.env strand)))
                  (braid.strands final-braid))
            final-braid))))))

;;; ---------------------------------------------------------------------
;;; The analyzer definition for PROGN.
;;; ---------------------------------------------------------------------

(define-analyzer progn (progn-form braid/strand)
  "Returns the braid resulting from analysis of PROGN-FORM in BRAID/STRAND."

  (let ((forms (if (eq (first progn-form) 'progn)
                   (cdr progn-form)
                   progn-form)))
    (mapc #'(lambda (form)
              (setf braid/strand (analyze-exp form braid/strand)))
          forms)
    braid/strand))

;;; ---------------------------------------------------------------------
;;; The analyzer definition for LIST.
;;; ---------------------------------------------------------------------

(define-analyzer LIST (list-form braid/strand)
  "Returns a braid representing the analysis of LIST-FORM in BRAID/STRAND.
   These lists are homogeneous."

  (with-strand (strand braid/strand)
    (cond ((cdr list-form)
           (with-arg-analyses ((fc env args)
```

```
                        (analyze-exps (cdr list-form) strand))
            (let ((error-braid (fresh-braid
                                (make-singleton-braid
                                  :fc fc
                                  :env env
                                  :result (make-generic-list))))
                  (list-element-analysis (reduce-unify args)))
              (cond (list-element-analysis
                     (make-singleton-braid
                       :fc fc
                       :env env
                       :result (make-type-op
                                 :name 'list
                                 :args `(,list-element-analysis))))
                    (t
                     (essie-warning "Non-Homogeneous List: ~s" list-form)
                     error-braid)))))
          (t
           (analysis->result (make-generic-list) strand)))))

;;; ---------------------------------------------------------------------------
;;; The analyzer definition for FUNCTION (i.e., #')
;;; ---------------------------------------------------------------------------

(define-analyzer function (fn-form braid/strand)
  "Returns a function analysis object."

  (arg-check braid/strand braid/strand-p
             fn-form form-p)

  (with-strand (strand braid/strand)
    (let ((fn-spec (cadr fn-form)))

      (cond ((symbolp fn-spec)
             ;;retrieve the function analysis from the environment.
             (let ((fn-analysis (or (get-analysis fn-spec)
                                    (get-analysis fn-spec
                                                  (strand.env strand)))))
               (if fn-analysis
                   (make-singleton-braid
                     :fc (strand.fc strand)
                     :env (strand.env strand)
                     :result fn-analysis)
                   (essie-error "Unrecognized symbol: ~s" fn-spec))))

            (t
             ;;lambda list form of function.
             (let* ((param-id-list (reverse (get-param-ids
```

```
                                    (second fn-spec))))
      (param-vars (mapcar #'(lambda (param)
                              (declare (ignore param))
                              (make-type-var))
                          param-id-list)))
;;first, add the parameters to the env
(push-env (strand.env strand) param-id-list param-vars
          :parameters)
(let* ((fn-braid (call-analyzer 'progn (cddr fn-spec)
                                strand))
       (processed-braid
         (process-defun-braid fn-braid fn-spec)))

  ;; pop the lambda's parameters off the strand's env.
  (make-singleton-braid
    :fc (strand.fc strand)
    :env (pop-env (strand.env strand))
    :result (make-func
              :s-exp fn-spec
              :braid processed-braid)))))))))
```

BIBLIOGRAPHY

[AS87] H. Abelson and G. Sussman. Lisp: A language for stratified design. AI memo 986, MIT, August 1987.

[ADWR86] G. Avrunin, L. Dillon, J. Wileden, and W. Riddle. Constrained expressions: Adding analysis capabilities to design methods for concurrent software systems. *IEEE Transactions on Software Engineering*, SE-12(2), Feb 1986.

[Bar87] D. Barstow. Artificial intelligence and software engineering. In *Proceedings of the Ninth International Conference on Software Engineering*, pages 200–211, March 1987.

[BSS84] D. Barstow, H. Shrobe, and E. Sandewall, editors. *Interactive Programming Environments.* McGraw-Hill, 1984.

[Bee88] R. Beer. The compile-time type inference and type checking of Common Lisp programs: a technical summary. TR-88-116, Case Western Reserve University, 1988.

[BDG+88] D. Bobrow, L. DeMichiel, R. Gabriel, S. Keene, G. Kiczales, and D. Moon. Common lisp object system specification X3JI3 document 88-2R. *SIGPLAN Newsletter*, September 1988.

[BS86] D. G. Bobrow and M. Stefik. Perspectives on AI programming. *Science*, February 1986.

[Boe76] B. Boehm. Software engineering. *IEEE Transactions on Software Engineering*, SE-25(12), Dec 1976.

[BTCGS89] V. Breazu-Tannen, T. Coquand, C. Gunter, and A. Scedrov. Inheritance and explicit coercion. In *Fourth Annual Symposium on Logic in Computer Science*, 1989.

[BMS80] R. Burstall, D. MacQueen, and D. Sanella. Hope: An experimental, applicative language. CSR-62-80, University of Edinburgh, 1980.

[Car87] L. Cardelli. Basic polymorphic type checking. *Science of Computer Programming*, 1987.

[CW86] L. Cardelli and P. Wegner. On understanding types, data abstraction, and polymorphism. *Computing Surveys*, 17(4), December 1986.

[CR81] L. Clarke and D. Richardson. Symbolic evaluation methods for program analysis. In *Program Flow Analysis: Theory and Applications*. Prentice-Hall, 1981.

[CS89] J. Connell and Linda Shafer. *Structured Rapid Prototyping: An evolutionary approach to software development.* Prentice-Hall, 1989.

[CGM86] D. Corkill, K. Gallagher, and K. Murray. GBB: A generic blackboard development system. *Proceedings of the National Conference on Artificial Intelligence*, Aug 1986.

[CC77] Patrick Cousot and Radhia Cousot. Automatic synthesis of optimal invariant assertions: Mathematical foundations. In *Proceedings of the ACM Symposium on Artificial Intelligence and Programming Languages*, volume 12(8), pages 1–12. SIGPLAN Notices, August 1977.

[DM82] L. Damas and R. Milner. Principal type schemes for functional programs. In *Conference Record of the Ninth ACM Symposium on Principles of Programming Languages*, 1982.

[FH88] A. Field and P. Harrison. Type inference systems and type checking. In *Functional Programming*. Addison-Wesley, 1988.

[For87] L. Ford. Artificial intelligence and software engineering: a tutorial introduction to their relationship. *Artificial Intelligence Review*, 1(4), 1987.

[FO76] L. Fosdick and L. Osterweil. Data flow analysis in software reliability. *ACM Computing Surveys*, 8, Sep 1976.

[Gan77] J. Gannon. An experimental evaluation of data type conventions. *Communications of the ACM*, 20(8), 1977.

[Gid84] R. V. Giddings. Accomodating uncertainty in software design. *Communications of the ACM*, 27(5):428–434, May 1984.

[Gol87] Adele Goldberg. Programmer as reader. *IEEE Software*, 4(5), September 1987.

[Goo81] J. Goodwin. Why programming environments need dynamic data types. *IEEE Transactions on Software Engineering*, SE-7(5), Sep 1981.

[GMW79] M. Gordon, A. Milner, and C. Wadsworth. *Edinburgh LCF.* Number 78 in Lecture Notes in Computer Science. Springer-Verlag, 1979.

[Gra89] J. Graver. *Type-checking and type-inference for object-oriented programming languages.* UILU-ENG-89-1758, University of Illinois, Aug 1989.

[Han87] P. Hancock. A type checker. In *The Implementation of Functional Programming Languages.* Prentice-Hall, 1987.

[Hec77] M. Hecht. *Flow Analysis of Computer Programs.* New-York: Elsevier North-Holland, 1977.

[HWA+90] P. Hudak, P. Wadler, Arvind, B. Boutel, J. Fairbairn, J. Fasel, K. Hammond, J. Hughes, T. Johnsson, D. Kieburtz, R. Nikhil, S. Jones, M. Reeve, D. Wise, and J. Young. *Report on the programming language Haskell: A non-strict, purely functional language*, Version 1.0 edition, Apr 1990.

[KU78] M. Kaplan and J. Ullman. A general scheme for the automatic inference of variable types. In *Conference Record of the Fifth Annual ACM Symposium on Principle of Programming Languages*, 1978.

[Kin76] J. King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, July 1976.

[Kni89] Kevin Knight. Unification: A multidisciplinary survey. *ACM Computing Surveys*, 21(1):93–124, March 1989.

[LR88] K. Lieberherr and A. Riel. Demeter: A CASE study of software growth through parameterized classes. *Journal of Object-Oriented Programming*, Aug 1988.

[LD89] M. Lowry and R. Duran. Knowledge-based software engineering. In A. Barr, P. Cohen, and E. Feigenbaum, editors, *The Handbook of Artificial Intelligence.* Addison-Wesley, 1989.

[Luq89] Luqi. Software evolution through rapid prototyping. *Computer*, 22(5):13–28, May 1989.

[Mac86] D. MacQueen. Using dependent types to express modular structure. In *Conference Record of the 13th ACM Symposium on Principles of Programming Languages*, 1986.

[MJ81a] D. McCracken and M. Jackson. A minority dissenting position. In *Systems Analysis and Design: A Foundation for the 1980's.* Elsevier Publishing Co., 1981.

[Mil78a] T. Miller. Type checking in an imperfect world. In *Conference Record of the Sixth Annual ACM Symposium on Principle of Programming Languages*, 1978.

[Mil78b] A. Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17(3), 1978.

[Mos85] J. Mostow. What is AI? and what does it have to do with software engineering? *IEEE Transactions on Software Engineering*, SE-11(11), Nov 1985.

[MJ81b] Steven S. Muchnick and Neil D. Jones, editors. *Program Flow Analysis: Theory and Applications.* Prentice-Hall, 1981.

[Nar88] K. Narayanaswamy. Static analysis-based program evolution support in the Common Lisp Framework. In *Proceedings of the 10th International Conference on Software Engineering*, 1988.

[OO90] K. Olender and L. Osterweil. Cecil: A sequencing constraint language for automatic static analysis generation. *IEEE Transactions on Software Engineering*, 16(3), Mar 1990.

[Ost81] L. Osterweil. Using data flow tools in software engineering. In *Program Flow Analysis: Theory and Applications.* Prentice-Hall, Inc., 1981.

[OF76] L. Osterweil and L. Fosdick. DAVE—A validation, error detection, and documentation system for FORTRAN programs. *Software—Practice and Experience*, 1976.

[Par86] D. Partridge. Engineering artificial intelligence software. *Artificial Intelligence Review*, 1(1):27–41, 1986.

[Par90] D. Partridge, editor. *Artificial Intelligence and Software Engineering.* Ablex, 1990. (Forthcoming).

[RS76] Charles Rich and Howard Shrobe. Initial report on a LISP programmers apprentice. AI-TR-354, MIT Artificial Intelligence Laboratory, 1976.

[RW86] C. Rich and R. Waters, editors. *Readings in Artificial Intelligence and Software Engineering.* Morgan-Kaufmann, 1986.

[Rob65] J. Robinson. A machine-oriented logic based upon the resolution principle. *Journal of the ACM*, 12(1), 1965.

[RP86] B. Ryder and M. Paull. Elimination algorithms for data flow analysis. *ACM Computing Surveys*, 18, Sep 1986.

[San77] E. Sandewall. Some observations on conceptual programming. In E.W. Elcock and D. Michie, editors, *Machine Intelligence 8.* John Wiley and Sons, 1977.

[San78] E. Sandewall. Programming in an interactive environment: The lisp experience. *Computing Surveys*, 10(1), 1978.

[She84] B. Sheil. Power tools for programmers. In D.R. Barstow, H.E. Shrobe, and E. Sandewall, editors, *Interactive Programming Environments.* McGraw Hill, Inc., 1984.

[Sim86] H. Simon. Whether software engineering needs to be artificially intelligent. *IEEE Transactions on Software Engineering*, SE-12(7), July 1986.

[Str67] C. Strachey. Fundamental concepts in programming languages. Lecture Notes for the International Summer School in Computer Programming, Copenhagen, 1967.

[Str83] R. Strom. Mechanisms for compile-time enforcement of security. In *Conference Record of the Tenth Annual Symposium on Principles of Programming Languages*, 1983.

[SB82] W. Swartout and R. Balzer. On the inevitable intertwining of specification and implementation. *Communications of the ACM*, July 1982.

[Te88a] M Tanik and D. Yun (eds). IEEE expert special issue on AIand software engineering, Winter 1988.

[Te88b] M. Tanik and D.Yun (eds). IEEE computer special issue on AI and SE, Nov 1988.

[Ten74] A. Tennenbaum. *Type Determination in Very High Level Languages.* PhD thesis, New York University, 1974.

[Tur85] D. Turner. Miranda—a non-strict functional language with polymorphic types. *Conference on Functional Programming Languages and Computer Architecture*, 1985.

[WB89] P. Wadler and S. Blott. How to make ad-hoc polymorphism less ad-hoc. *16th Annual Symposium on Principles of Programming Languages*, 1989.

[Weg75] Ben Wegbreit. Property extraction in well-founded property sets. *IEEE Transactions on Software Engineering*, SE-1(3):270–285, September 1975.

[Win73] T. Winograd. Breaking the complexity barrier (again). In *ACM-SIGIR-SIGPLAN Interface Meeting*, November 1973.

[Win79] T. Winograd. Beyond programming languages. *Communications of the ACM*, 22(7):391–401, August 1979.

[Wol89] W. Wolf. A practical comparison of two object-oriented languages. *IEEE Software*, Sep 1989.

[XW88] J. Xu and D. Warren. A theory of types and type inference for prolog. TR 88-15, Department of Computer Science, State University of New York at Stony Brook, 1988.