

**MULTIPLE STRONGLE TYPED EVALUATIN PHASES:
A PROGRAMMING LANGUAGE NOTION**

David Booth

**August 1986
CSD-860042**

UNIVERSITY OF CALIFORNIA

Los Angeles

**Multiple Strongly Typed Evaluation Phases:
A Programming Language Notion**

A dissertation submitted in partial satisfaction
of the requirements for the degree of
Doctor of Philosophy in Computer Science

by

David S. Booth

1985


© Copyright by

David S. Booth

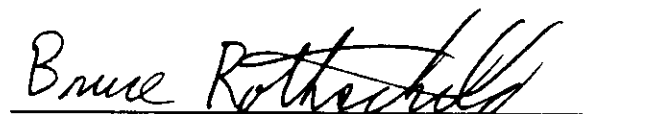
1985

The dissertation of David S. Booth is approved.


Daniel Berry


Gerald Estrin


Yiannis Moschovakis


Bruce Rothschild


David F. Martin, Committee Chair

University of California, Los Angeles

1985

Table of Contents

1. Introduction	1
1.1. Research Contribution	1
1.1.1. Ideas in This Research	2
1.1.1.1. An Abstract Data Type for Type-checked Program Fragments	2
1.1.1.2. One Machine Acts as Compiler and Runtime Machine	2
1.1.1.3. Multiple Strongly Typed Evaluation Phases	2
1.1.1.4. The Translator Does No Type Checking	3
1.1.1.5. One Machine Does Partial and Full Evaluation	3
1.1.1.6. Phase Compilation	3
1.1.1.7. An Unusual View of Abstract Data Types	4
1.2. Background to This Research	4
1.2.1. Two Models of Program Evaluation: Interpreted and Compiled	4
1.2.1.1. Definition: Runtime Type Errors	6
1.2.1.2. Definition: Strong Typing	6
1.2.2. The Purposes of Compiling	7
1.2.3. Problems with Traditional Compiled Languages	8
1.2.3.1. Lack of Programmer Control	8
1.2.3.2. Ad Hoc Notions	9
1.2.3.3. The Conflict Between "Strong Typing" and "Types as First-Class Values"	10
1.2.3.4. Different Mechanisms for Type Checking and Evaluation	10
1.3. This Research	11
1.3.1. Expressing versus Inferring	12
1.3.1.1. Advantages of Inference over Expression	13
1.3.1.2. Disadvantages of Inference as Opposed to Expression	14
1.3.1.3. The Gray Area Between Inference and Expression	15

1.4. Related Work	15
1.4.1. Pebble	15
1.4.2. Partial Evaluation	17
1.4.2.1. Definition of Partial Evaluation	17
1.4.2.2. Uses of Partial Evaluation	17
1.4.2.3. Comparing Phases and Partial Evaluation	19
1.4.3. Current Work by Gifford, Schooler, et al.	20
2. Programming Method	22
2.1. General Programming Method	22
2.1.1. Distinct Application and Implementation Languages	22
2.1.2. Programs Are Combined	24
2.1.3. Programs Are Instantiated	25
2.2. Specific Programming Method	25
2.2.1. ERT: Expression, Required-environment, Type	25
2.2.2. Valid ERTs	28
2.2.3. Interpreting the Specific Programming Method	28
2.2.4. A Command Interpreter	30
2.2.5. Environments	30
2.3. Motivating Example: General Purpose Sorting Function	31
2.3.1. The Desire for a General-Purpose Sorting Function	32
2.3.2. A Sorting Function Generator	33
2.3.3. Explicit Generation vs. Partial Evaluation	33
2.3.4. Phases Used	34
2.3.5. Generalizing Further	34
3. The Conceptual Model of Multiple Phases	37
3.1. Arriving at Phases by Extending Compiletime	37
3.1.1. Generalizing Compiletime	39
3.1.2. Generalizing Pre-compiletime, And So On . . .	40
3.2. Conceptual Model of Multiple Phases	42
3.3. Interpreting the Conceptual Model	42
3.3.1. Properties of the Conceptual Model	44
3.3.2. Resolving the Conflict Between "Strong Typing" and "Types as First-Class Values"	45
3.3.3. The Paradox of Strong Typing Without Prior Type Checking	46
3.3.4. All Expressions Start Out Type ERT	47
3.4. Assigning Computations to Phases	48
3.5. How Many Phases Are Required?	49

4. Static Determination of Phases	51
4.1. The Static-Phi Language	51
4.1.1. Conventional Static-Phi Language Constructs	52
4.1.2. Normal Runtime Phase	53
4.1.3. Some Unusual Constructs	54
4.1.4. Examples	55
4.1.4.1. F Twice	55
4.1.4.2. Identity Abstraction	56
4.1.4.3. Identity Application	56
4.1.4.4. Function Abstraction	56
4.1.4.5. Function Application	56
4.1.4.6. Higher Order Function Abstraction	56
4.1.4.7. Higher Order Function Application	57
4.1.4.8. Identity-Function Type Abstraction	57
4.1.4.9. Identity-Function Type Application	58
4.1.4.10. General Type Abstraction	58
4.1.4.11. General Type Application	59
4.1.4.12. Macro Abstraction	59
4.1.4.13. Macro Application	59
4.2. The Static-IL Language	60
4.2.1. Lexical Scoping, ERTs, and Macros	61
4.2.2. Conventional Lambda Calculus Operations	63
4.2.3. Some Unusual Operations	64
4.2.3.1. $(\text{deep-const } c \ t \ n)$	65
4.2.3.2. $(\text{check-funtype } \text{expr}_d \ \text{expr}_r \ n)$	65
4.2.3.3. $(\text{check-lambda } id \ \text{expr}_d \ \text{expr}_r \ \text{expr}_{\text{body}})$	67
4.2.3.4. $(\text{check-check-lambda } id \ \text{expr}_d \ \text{expr}_r \ \text{expr}_{\text{body}} \ n)$	68
4.2.3.5. $(\text{check-apply } \text{expr}_f \ \text{expr}_x)$	70
4.2.4. Efficiency of Static-IL	71
4.3. Environments for Static-IL Programs	72
4.3.1. Static-Phi Program X	72
4.3.2. Definition: Default ERT	73
4.3.3. The Purpose of Default ERTs	73
4.3.4. Fixing the Type of an Identifier	73
4.3.5. Fixing an Identifier as a Function Type	74
4.3.6. Fixing an Identifier as a Macro	74
4.3.7. Fixing the Value of an Identifier	75
4.3.8. Other Possibilities	75

4.3.9. What Values to Supply in What Phases	75
4.4. Examples of Translation and Evaluation	76
4.4.1. F Twice	78
4.4.2. Identity Abstraction	79
4.4.3. Identity Application	80
4.4.4. Function Abstraction	83
4.4.5. Function Application	85
4.4.6. Higher Order Function Abstraction	87
4.4.7. Higher Order Function Application	89
4.4.8. Identity-Function Type Abstraction	91
4.4.9. Identity-Function Type Application	92
4.4.10. General Type Abstraction	95
4.4.11. General Type Application	97
4.4.12. Macro Abstraction	100
4.4.13. Macro Application	102
5. Using Phases for Partial Evaluation	105
5.1. The Dynamic-Phi Language	105
5.2. The Dynamic-IL Language	106
5.2.1. Conventional Lambda Calculus Operations	106
5.2.2. Other Operations	107
5.2.2.1. (hold <i>t expr</i>)	107
5.2.2.2. (check-funtype <i>expr_d expr_r</i>)	108
5.2.2.3. (check-lambda <i>id expr_d expr_r expr_{body}</i>)	109
5.2.2.4. (check-check-lambda <i>id expr_d expr_r expr_{body}</i>)	110
5.2.2.5. (check-apply <i>expr_f expr_x</i>)	110
5.3. Problems in Implementing Check-Lambda	110
5.3.1. Evaluating the Body Twice	114
5.3.2. Evaluating with Both Choices at Once	114
5.3.3. Using an Extra Environment Variable	114
5.3.4. The Root of the Problem	115
5.4. A Strongly Typed Phase Compiler	115
6. Remarks About Other Language Notions	117
6.1. Abstract Data Types	117
6.1.1. Four Essential Functions	118
6.1.2. Type Values: < <i>tag,value</i> > Pairs	120
6.1.3. Type-of and Tag-of	121
6.1.4. Implementations of the Four Essential Functions	121
6.1.5. Defining a New Abstract Data Type	122

6.2. Dependent Types	124
6.3. Type Checking Recursive Functions	125
7. Conclusions and Future Work	127
7.1. Conclusions	127
7.2. Subjects for Further Study	129
7.2.1. Developing a Practical Language Based on Static-Phi and Static-IL	129
7.2.2. Using Phases for Partial Evaluation	129
7.2.3. Constructing and Maintaining Environments	129
7.2.4. Determining the Source of a Bug	130
7.2.5. Universal Polymorphism	130
7.2.6. Inferring Types	131
7.2.7. Recursive Types	132
7.2.8. ERT Subtypes	134
7.2.9. Statically Inferred Phases	135
References	136
Appendix A. Formal Semantics of Static-Phi and Static-IL	141

List of Figures

Figure 1-1:	Two Models of Program Evaluation	5
Figure 2-1:	General Programming Method	23
Figure 2-2:	Specific Programming Method	26
Figure 2-3:	Phases of Sorting Example	36
Figure 3-1:	Traditional Compiletime and Runtime	38
Figure 3-2:	Pre-compiletime Phase Added	41
Figure 3-3:	Conceptual Model of Multiple Phases	43
Figure 6-1:	Implementing Abstract Data Types	123

Acknowledgements

At UCLA, thanks to David Martin for agreeing to be my advisor on this research even before I could fully explain what it was all about, and for following through as he found out. Thanks to Gerald Estrin for running the doctoral seminar that provided a much-needed forum for working out the presentation of these ideas, and for spending much more time than his role would normally warrant reading drafts of this work. And thanks to those students in the doctoral seminar who courteously sat through some rough early presentations of this work and provided valuable feedback.

At USC/ISI, thanks first to Vance Tyree for bringing me to the MOSIS group here at ISI and establishing me in this very supportive and much-appreciated environment. Thanks to George Lewicki for continuing to support me on the MOSIS group as this dissertation was written. Thanks to Sheila Coyazo for her excellent editorial help, and thanks to the others who have made working here at ISI such a pleasure for the past four years.

VITA

- 1979 B.S. *Magna cum laude*, University of New Hampshire
- 1981–1982 Teaching Assistant, University of California, Los Angeles
- 1982 Instructor, University of California, Los Angeles,
Engineering Extension
- 1983 M.S., University of California, Los Angeles
- 1982–1984 Graduate Research Assistant, Information Sciences Institute
of the University of Southern California
- 1984–1985 Systems Analyst, Information Sciences Institute of the
University of Southern California

ABSTRACT OF THE DISSERTATION

Multiple Strongly Typed Evaluation Phases:

A Programming Language Notion

by

David S. Booth

Doctor of Philosophy in Computer Science

University of California, Los Angeles, 1985

Professor David F. Martin, Committee Chair

This work introduces the programming language notion of *multiple strongly typed evaluation phases*, or simply, *phases*. In general, a program might be executed through several phases. Each phase requires its own input, and acts as compiletime relative to the next phase, or runtime relative to the previous phase. Thus, each phase is the execution of a program, and may play the role of compiletime or runtime.

The notion of phases offers a framework for understanding compiled strongly typed languages, and works toward an improved, strongly typed language basis for reusable software. The research shows how types can be manipulated as first-class values, and notions of compiletime and runtime can be unified, without sacrificing strong typing (compiletime type checking) or runtime speed. Type checking and expression evaluation are performed using the same evaluation mechanism.

The apparent conflict of allowing types as first-class values, yet enforcing compiletime type checking, is resolved by the notion of multiple phases: though types may be manipulated as first-class values during one phase, the computed type values become invariants for the next phase.

We demonstrate the notion of phases by defining a sample source language, Phi, which looks like a typed lambda calculus; an object language, IL, which is syntactically similar to an untyped lambda calculus, but is strongly typed; an associated IL Machine that interprets IL programs; and a translator for converting Phi programs to IL programs. Strong typing is guaranteed in spite of the fact that the Phi translator does no type checking. We also discuss how phases might be used to efficiently perform partial evaluation.

A phase, i , is the execution of an IL program, p_i . The result may be another IL program, p_{i+1} , to be executed in phase $i+1$, or it may be the desired final answer. Phase i acts as compiletime for phase $i+1$, doing all type checking necessary to guarantee that program p_{i+1} is free of runtime type errors. During phase i , program p_i can manipulate types as first-class values; in general these computed types will be invariants of the next phase.

Chapter 1

Introduction

1.1. Research Contribution

This dissertation describes a programming language notion -- *phases* -- and an associated programming method. Its contribution is both practical and academic: it takes a small step toward providing a strongly typed language basis for more reusable software; and it provides a more general, unified view of certain notions in programming languages and methodology, including compiletime and runtime.

This work is not advocating any *particular* programming language or method. The main intent of this dissertation is to expose the essence of multiple strongly typed evaluation phases, without encumbering the reader with extraneous details or tangential issues. We illustrate the essential ideas by defining some pedagogical languages based on the Lambda Calculus [Barendregt 84]: Phi and IL. As of this writing, two versions of these languages have been implemented and tested.

1.1.1. Ideas in This Research

It is often difficult, in reading research reports, to distill the important ideas being advocated from the mundane details of the particular system described. Outlined below are what the author considers to be the most interesting ideas embodied in this research.

1.1.1.1. An Abstract Data Type for Type-checked Program Fragments

A particular abstract data type (the data type ERT) is defined for constructing and manipulating type-checked programs. This allows program fragments to be securely manipulated as data, and thus allows compiletime operations to be treated in the same manner as runtime operations. The primitive operations implementing this data type ensure that every program constructed in this way is syntactically correct and strongly typed. (Section 2.2.1.)

1.1.1.2. One Machine Acts as Compiler and Runtime Machine

Notions of compiletime and runtime are unified: compiletime operations are generalized and become a superset of runtime operations. A single abstract machine can do both efficiently. (Section 3.1.2.)

1.1.1.3. Multiple Strongly Typed Evaluation Phases

Each phase is the execution of a program on the abstract machine. The result of each phase may be the final answer or another type-checked program. Each phase type checks and generates the program for the next phase. Types may be manipulated as first-class values during any phase; they become invariants for the next phase. "Compiletime" and "runtime" thus become relative terms. (Sections 3.2 and 3.3.)

1.1.1.4. The Translator Does No Type Checking

Given a program in the source language, the translator can produce a strongly typed program in the implementation language without doing any type checking. That is, the translator does no type checking, but the resulting program is guaranteed free of runtime type errors. This fact may at first sound contradictory; it is explained in Section 3.3.3.

1.1.1.5. One Machine Does Partial and Full Evaluation

A single abstract machine can efficiently do both partial evaluation and full evaluation. (Chapter 5.)

1.1.1.6. Phase Compilation

Chapter 5 discusses how phases might be used for partial evaluation for a strongly typed language. Partial evaluation is often slow, but it might be made more efficient by using two steps: *phase compilation* and *phase evaluation*.

Given a list of the free variables to be given fixed values, a *phase compiler* would prepare a program for phase evaluation, which will achieve the effect of efficient partial/full evaluation. The program would first be "phase compiled," using a list of the free variables -- and their types -- to be instantiated. Efficient partial/full evaluation would then be performed by executing this "phase compiled" program using phase evaluation. (Section 5.4.)

1.1.1.7. An Unusual View of Abstract Data Types

Since types and code are first-class values, our view of Abstract Data Types (ADTs) is in terms of what primitive *functions* are necessary in order to support user-defined ADTs. Operationally, one needs these functions in order to convert between the domains of the *abstraction* and the *representation*. However, they can be ordinary functions rather than special language constructs. (Section 6.1.)

1.2. Background to This Research

Programming language experts should read the definitions of "runtime type errors" and "strong typing" in Sections 1.2.1.1 and 1.2.1.2, but may otherwise wish to skip to Section 1.3, which describes this research.

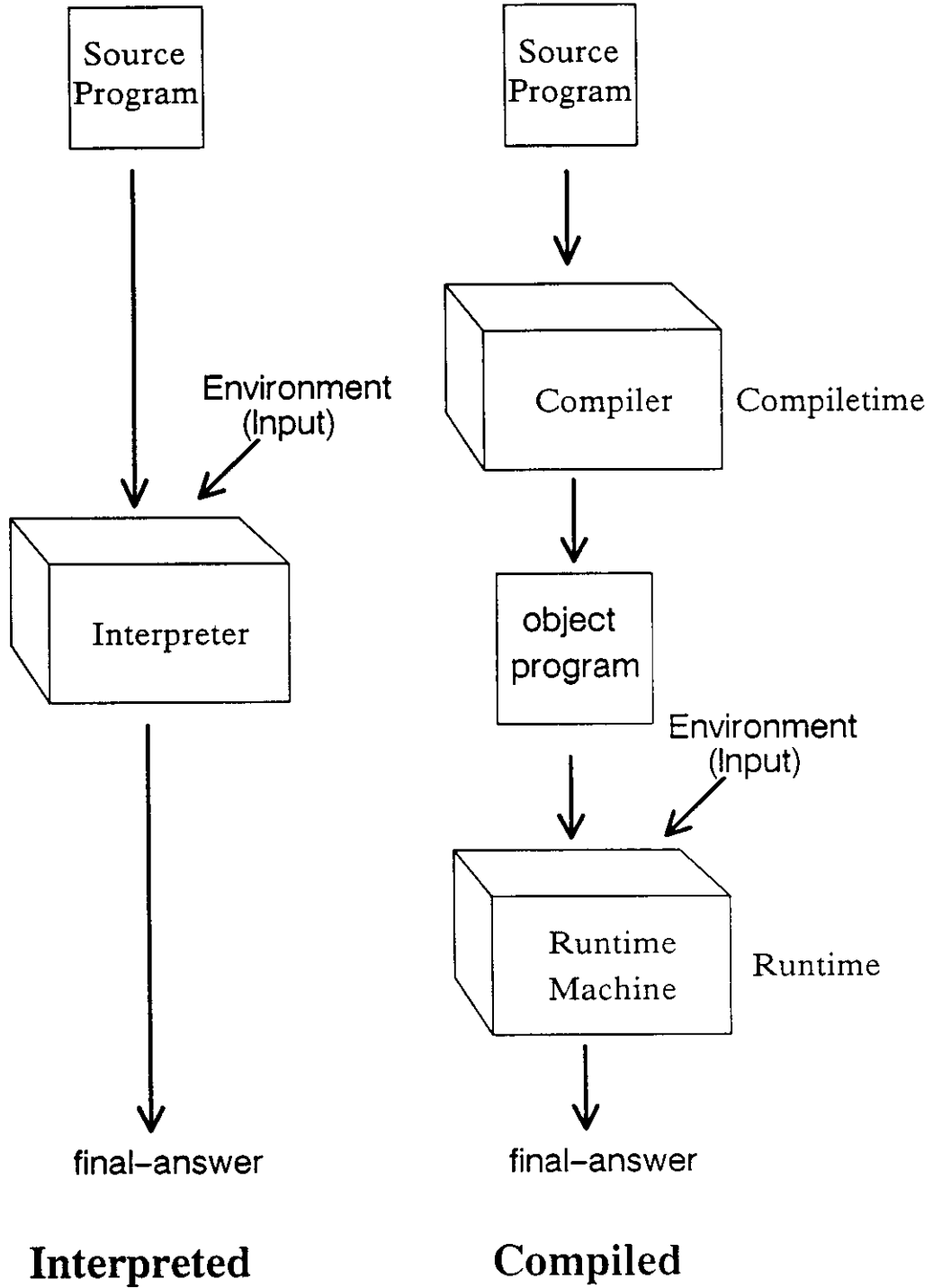
1.2.1. Two Models of Program Evaluation: Interpreted and Compiled

Figure 1-1 shows two models of how a source program written in some language L might be evaluated.

In the *interpreted* case, the program is given directly to an interpreter. A program generally also needs a specified environment, which might include values for the program's input variables and definitions of some standard functions. The interpreter runs the program with the given environment and produces the desired result of the computation -- the *final answer*, which might be some number, a character string, or a more complex object such as a file.

In the *compiled* case, the source program is first translated, by a compiler, into an object program in some other language L'; this step is called

**Figure 1-1:
Two Models of Program Evaluation**



compiletime. An L' interpreter then runs the object program with the desired environment to produce the final answer; this step is called *runtime*. The object program may be stored and run repeatedly using different environments or inputs, without re-translating the source program.

This work concerns the compiled, rather than the interpreted, model.

1.2.1.1. Definition: Runtime Type Errors

Suppose the source program contains a mistake, causing the L interpreter (in the interpreted case) or the L' interpreter (in the compiled case) to try to apply some erroneous operation, such as multiplying two character strings. It may be detected by the interpreter, and an error message issued, or it may not be detected, in which case the result of the computation will be garbage. In either case, it is called a **runtime type error** to distinguish it from any errors that the compiler might issue before the program is executed.

1.2.1.2. Definition: Strong Typing

If the source program contains adequate information about the types of values to be computed, the compiler can ensure that the generated object program will be free of runtime type errors. **Strong typing** means providing

an *a priori*, or compiletime, guarantee against runtime type errors.¹

This work concerns only languages providing strong typing.

1.2.2. The Purposes of Compiling

There are two basic advantages to compiling the source program, as opposed to interpreting it directly: type security and efficiency.

Type security Because the source language is strongly typed, the compiler can provide an *a priori* guarantee that no runtime type errors will occur when the object program is executed on the implementation machine. This provides an assurance that the program is at least partially correct, without executing the program.

Efficiency A compiler can improve a program's runtime efficiency in three ways: by computing constant expressions at compiletime; by selecting optimal object program code, based on values and types known at compiletime; and by translating the program into a language inherently more

¹The question sometimes arises: Is division by zero considered a runtime type error? What about an array index out of bounds? Or an attempt to read beyond the end of the input?

Paul Eggert [Eggert 81] has shown that it is possible to define the type system securely enough that such runtime errors are not possible. For example, one can define a type **non-zero-integers** that includes all integers except zero, and another type **possibly-zero-integers** that includes all integers. Only values of the type **non-zero-integers** would be allowed as divisors, and (for example) subtraction of two **non-zero-integers** would yield a result of type **possibly-zero-integers**. To convert a value of type **possibly-zero-integers** to a value of type **non-zero-integers**, one must use a special case-conformity clause, placing the detection of a zero value under explicit program control. The type system can similarly be defined in such a way that array-index-out-of-bounds and other such errors are not possible. Although many languages that purport to be strongly typed, such as Pascal, allow such loopholes in the type system, this work assumes that the type system is defined securely enough that such runtime errors are not possible.

efficient for the implementation machine to execute.²

1.2.3. Problems with Traditional Compiled Languages

The benefits of compiling are well established, and languages specifically designed to be compiled -- *compiled languages* -- are common. In spite of these advantages, there are some problems with traditional compiled languages.

1.2.3.1. Lack of Programmer Control

Inherently, the compiler must know a great deal about the source program and the types of values being manipulated in order to produce an efficient, type-checked object program. However, the programmer generally does not have access to much of this compiletime information.

For example, in Pascal, there is no way to ask for the size of an array or for the first value of an enumerated type.³ Certainly the compiler has this information, but the programmer has no way of accessing it.

²This third method of improving efficiency will be ignored when we generalize compiletime to arrive at the notion of phases. However, the idea of translating to a more efficient language is not incompatible with the notion of phases. Instead of executing an Implementation Language program directly, we could first translate it to another more efficiently executed language.

³An enumerated type is a type for which all values are explicitly listed, for example, `type color = (red, green, blue)`.

1.2.3.2. Ad Hoc Notions

It is easy to see similarities between the kinds of operations performed by the compiler at compiletime, and the operations performed under program control at runtime. In spite of the conceptual similarities, compiletime notions tend to be ad hoc. For example, the shortcomings of Pascal mentioned above were addressed in Ada⁴ by supplying *attribute* operations, which ask for an array's size or an enumerated type's first value. The Ada Reference Manual [Ada 82] defines 48 such attributes! Some of these attributes are computable at compiletime and some are not.

Type expressions are usually treated very differently from other -- conventional -- expressions, such as numeric expressions. In fact, they usually have different syntactic rules. Consider Pascal. One can define a variable *x* to be some user-defined type *t*:

```
var x: t;    { t is some user-defined type }
```

Or one can declare *x* using an array type expression involving *t*:

```
var x: array [1..20] of t;
```

However, one cannot compute an arbitrary function of *t*:

```
var x: f(t); { Illegal }
```

To various extents, some languages, such as Donahue's Extended Lambda Calculus [Donahue 79], Russell [Boehm 80], EL1 [Wegbreit 74] and Pebble [Burstall 84], do treat types as *first-class values*; that is, one may use type variables and write functions and expressions involving types. However, these languages tend to syntactically separate type expressions from normal expressions, restrict the kinds of computations allowed on

⁴Ada is a registered trademark of the U.S. Government. Ada Joint Program Office.

types to ensure that the type values are statically computable, or forego strong typing and use runtime type checking. Pebble's treatment of types is general and uniform in these respects, but it does treat one aspect of types differently, as mentioned in Section 1.2.3.4.

1.2.3.3. The Conflict Between "Strong Typing" and "Types as First-Class Values"

The motivation for allowing types as first-class values is clear: the abilities to parameterize by types, use arbitrary algorithms to construct new types, and make decisions based on types, would support more reusable software. Similarly, the benefits of strong typing are well established: type security and efficiency.

Unfortunately, there is an inherent conflict between allowing types as first class values and the desire for strong typing. Basically, strong typing requires that the type of every expression be known before runtime. However, allowing types as first-class values means that types may involve arbitrary expressions, use variables, invoke functions, depend on input, etc.

1.2.3.4. Different Mechanisms for Type Checking and Evaluation

Type checking is similar to program evaluation. The similarity is readily apparent when one compares a typical language's semantic rules for type checking with its semantic rules for evaluation: both draw conclusions about an expression's value or type based on the values or types of the expression's subexpressions, and both follow lexical scoping rules for identifiers.

Nonetheless, strongly typed languages have invariably defined separate

mechanisms for type checking and program evaluation. For example, even in Burstall and Lampson's Pebble [Burstall 84], though type checking involves evaluation, a different mechanism is used for type checking than for evaluation. This is shown clearly in Table 6, Section 5.3 of Pebble [Burstall 84], where the type checking rules are separated from the evaluation rules to form what is essentially a different machine. (The rules are separated to demonstrate the distinction between the act of type checking and the act of evaluation.) Both sets of rules apply to the same language constructs, but they are applied at different times, depending on whether the program is being type checked or executed.

1.3. This Research

Can types and code be manipulated effectively under programmer control during compiletime, while retaining strong typing? Can compiletime notions such as type checking be unified with runtime notions?

The answer is "Yes." The language notion of *multiple strongly typed evaluation phases*⁵ unifies compiletime and runtime, and allows types and code to be manipulated as first-class values, while retaining strong typing. Types, manipulated as first-class values in one phase, become invariants of the next phase, as explained in Section 3.3.2. Phases might also be used to perform partial evaluation, as discussed in Section 5. The purpose of this work is to explore and introduce the notion of multiple strongly typed

⁵The term *phases* is often used in this work instead of the longer, more descriptive term *multiple strongly typed evaluation phases*.

evaluation phases.⁶

Our particular approach to type checking was motivated by certain key biases:

- A firm belief in **strong typing**, that is, in providing an *a priori* guarantee that a program is free of any possible runtime type errors.
- A desire to **unify** the notions of **compiletime** and **runtime**.
- An orientation toward explicit **programmer expression** rather than inference performed by the language implementation. These orientations are contrasted in Section 1.3.1.
- A desire to support the general programming method described in Section 2.1.

1.3.1. Expressing versus Inferring

Programming languages are designed under two competing orientations: the programmer can **express** information, or the language implementation can **infer** the information. The Phi language described in Section 4.1 is strongly oriented toward *expressing* rather than *inferring*. This section explains this choice and the differences between the two orientations.

⁶This work was approached a little differently than most doctoral research. Rather than first carefully defining a problem and then seeking a solution, we pursued an interesting idea and developed it to see how it might be useful. This unusual approach is risky, because there is less assurance of a useful outcome, and it places a greater burden on the researcher for scholarly review and integration of related work. Nonetheless, this approach should be encouraged much more. The traditional approach of defining a problem and then seeking a solution is contrary to creativity, because every problem definition presupposes a certain view of the world. The most interesting and innovative developments are those that change one's view of the world, making problems *irrelevant* instead of solving them.

For example, rather than requiring the compiler to *infer* the type of an expression from its context, in Phi the type is simply *computed* as any other computation. Another example of this distinction is that type checking polymorphic functions in ML [Gordon 79] involves unification, a process of pattern matching to find the most general type solution. If the same kind of polymorphic functions were offered in a language oriented toward programmer *expression*, the programmer would have the responsibility of *expressing* the desired type solution, and the language should provide useful type operations to make this easy. This is like the difference between proof *checking* and proof *discovery*.

1.3.1.1. Advantages of Inference over Expression

The main argument for having the compiler infer whatever it can is that it reduces the burden on the programmer. This is a good argument, but it is not prima-facie evidence that compiler inference is preferable to language expressiveness. It does, however, point out that *ease* of expression is very important. Concise syntactic constructs and libraries of reusable components should be provided to make expression easy.

Another argument for having the compiler infer information is that the inferences are assured correct (assuming that the compiler is correct, and that the programmer understands the inferences). If the programmer is given the responsibility of computing the types of expressions, for example, it is conceivable that the programmer would occasionally make a mistake and compute the wrong type, thus allowing an operation to be applied erroneously. In this case (to ensure strong typing), if the programmer is allowed to compute types arbitrarily, it is clear that the compiler must have some way of verifying that any computed types are in fact legal.

1.3.1.2. Disadvantages of Inference as Opposed to Expression

One disadvantage of relying on the compiler to infer information is that the compiler must be more complex. Thus, compilation may involve such tasks as unification or solving systems of simultaneous equations.

Perhaps the most important disadvantage, though, is that the programmer may want to express things that the compiler is not capable of inferring. This may be viewed as both a theoretical and a practical problem. As a simple example of the theoretical difficulty, suppose that every expression in the language must be guaranteed to halt, and that this is considered part of the expression's type correctness. The halting problem shows that this is theoretically impossible for the compiler to algorithmically determine, however, a compiler could much more easily verify a proof supplied by the programmer. As another example of the theoretical difficulty, Coppo [Coppo 80] asserts that when the type system of ML [Gordon 79] is extended, the question of whether a term possesses a type becomes only "semi-decidable".

The practical difficulty is that the compiler may not be smart enough to allow constructs that the programmer may wish to express. And unfortunately, making the compiler smarter generally makes it more complex.

1.3.1.3. The Gray Area Between Inference and Expression

There is no rigid distinction between inference and expression. For example, under the expressive orientation, a library routine implementing an inference engine could be provided. Or conversely, a language implementation's inference rules could simulate expression evaluation. Language processors generally contain elements of both inference and expression.

The work presented here is based on a strong bias toward expression, tempered with the compiletime checks necessary to ensure that any computed type values are legal. We do not intend to argue that expression is unequivocally *better* than inference. We are simply pointing out the importance of this orientation with respect to this work.

1.4. Related Work

1.4.1. Pebble

The Pebble language, by Burstall and Lampson [Burstall 84], uniformly allows types, bindings, and declarations as first-class values. Pebble's **bindings** are name-value pairs; they are essentially environments. Giving explicit access to bindings as first-class values makes it easy to build and access libraries or modules of reusable functions or other values under programmer control. Pebble's **declarations** are the types of bindings.

For simplicity, and to focus attention only on the notion of phases, in Phi we do not provide bindings and declarations as first-class values, though they would be very interesting to add. The idea fits our general philosophy

perfectly.⁷

Pebble also provides dependent types (see Section 6.2), though our Phi language does not. The need for them in Phi is somewhat reduced by the notion of multiple phases; this is discussed in Section 6.2.

Pebble deals with *language* ideas, whereas the notion of strongly typed evaluation phases might be more accurately characterized as a *language implementation* idea. As such, Pebble's semantic rules have no rigid separation between evaluation stages representing compiletime and runtime. However, Pebble's type checking and evaluation rules can be separated to provide static type checking. This separation essentially leads to different machines (that are applied to the same program) for doing type checking and evaluation. In contrast, our work provides a single machine that performs both roles of type checking and evaluation, depending on the expressions in the program. To clarify this distinction, in Pebble, whether a program is being type checked or evaluated depends on the set of rules applied -- it does not depend on the program itself. Whereas in our work, the syntax of the program determines whether our single type-checking-and-evaluation machine will do type checking or conventional evaluation.

⁷In fact, the first implementation of phases did treat bindings and declarations as first-class values.

1.4.2. Partial Evaluation

Partial evaluation is variously also known as *symbolic evaluation*, *partial execution*, *symbolic execution*, or *mixed computation*. Ershov [Ershov 77a] [Ershov 82] has probably been its main proponent.

1.4.2.1. Definition of Partial Evaluation

Partial evaluation reduces one program to another equivalent program in which some parts of the first program have been evaluated or simplified. For example, the expression $a + b + 2 * 3$ might be reduced to the equivalent expression $a + b + 6$. Or, if a value of 5 is provided for variable b , expression $a + b + 2 * 3$ might be partially evaluated to $a + 11$.

In general, if one or more of a program's input parameters are constant, the program may be *partially evaluated* to produce a new, more efficient program by taking advantage of those known constant values.

1.4.2.2. Uses of Partial Evaluation

Partial evaluation has mainly been used as a flexible mechanism for specializing programs. The purpose has generally been to produce a more efficient resulting program -- part of the computation has been done already. This efficiency motive is one of the two basic reasons for compiling programs as opposed to interpreting them directly.⁸ However, the advantage of partial evaluation over compilation is its flexibility -- any subset of a program's free variables (or inputs) can be fixed by supplying particular values for them. Gifford, Schooler, et al. [Schooler 84] are also working on using partial evaluation to perform type checking; this correctness motive is the other basic reason for compiling.

⁸Section 1.2.2 outlines the basic purposes of compiling.

Partial evaluation is also useful in separating notation from data representation. For example, in Pascal, the syntax for accessing data is tied to the representation of the data, making it difficult to change data representations. The programmer must choose between representing some data as a function or in a record, a linked list, or an array, and the syntax for accessing the data reflects this choice:

- a(b) Function invocation.
- a.b Accessing a component of a record.
- a↑.b Accessing through a pointer variable.
- a[b] Array subscripting.

Function invocation is the most general case, because any kind of data structure can be hidden inside the function body.⁹ Why shouldn't the programmer always hide the data structure inside a function? The answer is the traditional high cost of function invocation. But using partial evaluation, the function call can be avoided by beta-expanding¹⁰ (also called *beta-reducing*) the function call in-line, thus eliminating the performance justification for using specialized notation. Beta-expanding recursive functions can be a problem in general, but since (at the moment) we are simply discussing the possibility of hiding data structure access inside of function calls, recursive functions are not an issue here.

⁹Actually, in Pascal, only scalar types can be returned by a function. However, other languages do not have this restriction.

¹⁰*Beta expansion or beta reduction* replaces a function call with the function's body, having substituted actual parameters for formal parameters in the body. Care must be taken to preserve the properties of lexical scoping. Beta expansion is similar to macro expansion, except that macro expansion does not always guarantee that the properties of lexical scoping are preserved.

1.4.2.3. Comparing Phases and Partial Evaluation

As developed in Chapter 4, phase evaluation differs from partial evaluation in two important ways: (1) a program's various phases are explicitly indicated in the application program, and (2) program fragments can be manipulated as first-class values of an abstract data type (the data type ERT). The latter difference gives a macro-like capability, and the primitive operations that implement the abstract data type ensure that all generated programs are type correct.

The development in Chapter 5 shows how modifying and restricting phases might result in a system that essentially performs partial evaluation. Section 5.4 proposes a "phase compiler" approach that is analogous to the "compiled generation" approach of Beckman, et al. [Beckman 76], but ours applies to strongly typed languages, whereas theirs applied to the untyped language LISP [McCarthy 66]. This approach allows one abstract machine to efficiently perform both "partial" evaluation and "full" evaluation.

If phases were adapted to perform partial evaluation as discussed in Chapter 5, the most important remaining differences between phase evaluation and partial evaluation would be that: (1) phase evaluation syntactically distinguishes between those portions of a program that are being "partially" evaluated and those that are being "fully" evaluated, thus allowing the phase evaluator to perform both "partial" and "full" evaluation efficiently; and (2) under phase evaluation, a program's result type is always known before the program is evaluated.

1.4.3. Current Work by Gifford, Schooler, et al.

Gifford, Schooler, et al. apparently assume a similar general programming method to ours (described in Section 2.1). Their "kernel" language, the Imagine Base Language (IBL), corresponds to our Implementation Language (IL). Their programming method also assumes a partial evaluator, compilers, and interpreters, whereas ours includes a single Implementation Language Machine; our programming method makes explicit the operation of combining programs to form new programs, whereas theirs does not.

Their approach to providing an extensible, yet efficient, language is based on partial evaluation: specially defined forms can be converted to simpler, more efficient forms by partial evaluation. From Schooler [Schooler 84]:

Our proposed methodology is a generalization of the Russell [Boehm 80] and EL1 [Wegbreit 74] techniques: all [language] extensions are implemented in the language, allowing full user access to the extension mechanism. In addition, partial evaluation will be used to optimize the code to the point where using the user-defined extension mechanisms is essentially free in terms of runtime performance.

Gifford, Schooler, et al. also use their "front end" translators to insert assertions into the kernel language (IBL) code, and use the partial evaluator to compute as many of these assertions as possible. Since type checking is handled by inserting assertions about types, they thus provide compiletime type checking where possible and runtime type checking where necessary. Again from Schooler [Schooler 84]:

The code which the partial evaluator acts on will be generated by syntactic transforms from surface language constructs. The generated code will preserve all user-specified side-effects but will also include applicative constructs for type checking, etc.

Finally, since Gifford, Schooler, et al. are using partial evaluation, the comments on partial evaluation given in Section 1.4.2.2 apply to their work as well.

Chapter 2

Programming Method

This chapter discusses an assumed programming method. This programming method is very simple and rudimentary, and is not the focus of the research. It is included only to provide the necessary framework for discussing the main thesis of this work: the notion of phases.

The reader wishing to skim this chapter must be sure not to skip over Section 2.2.1, which defines ERTs, and is essential to subsequent chapters.

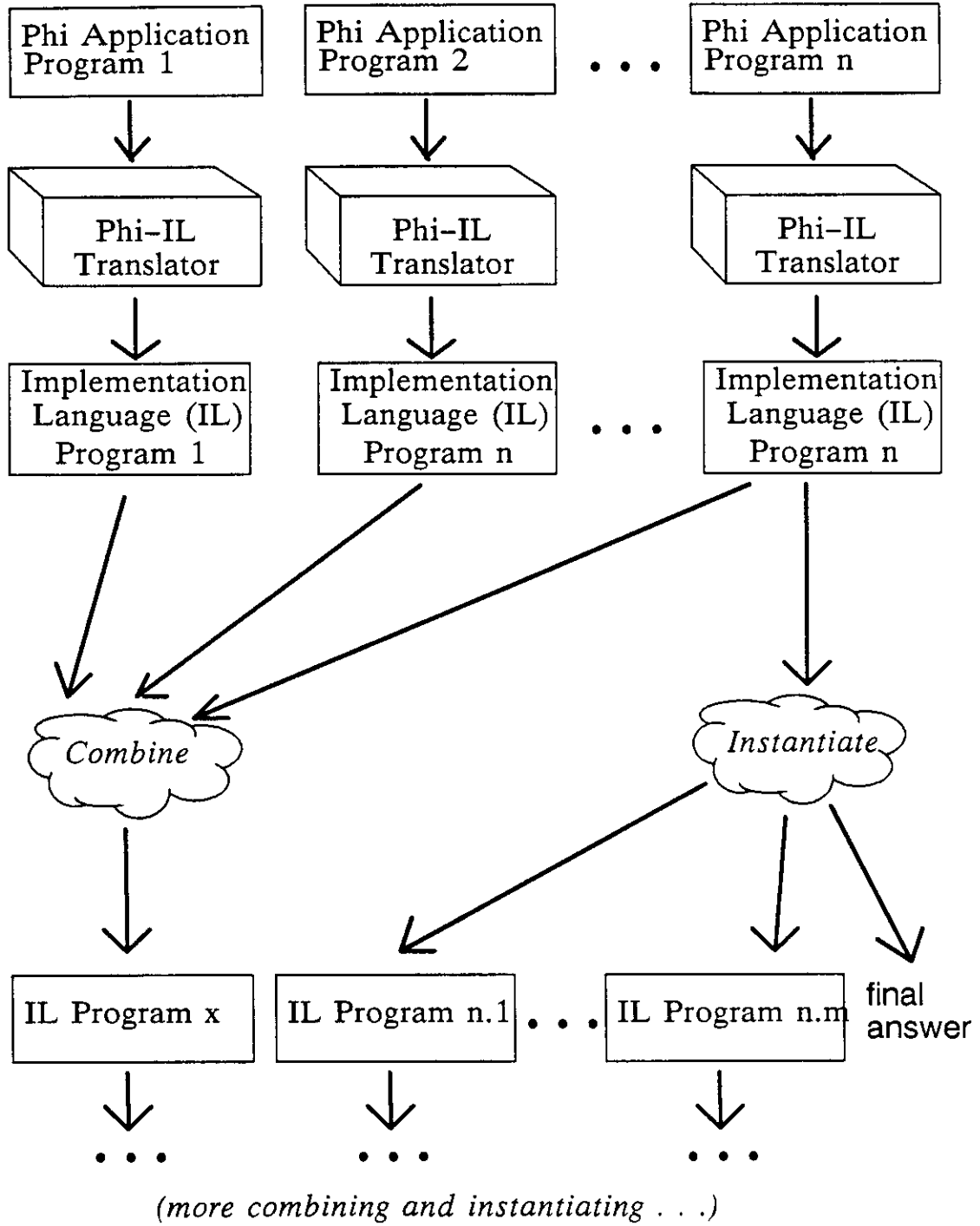
2.1. General Programming Method

The general programming method shown in Figure 2-1 illustrates how programs (or program fragments) may be used to create other programs. There are three essential aspects, described in the following sections.

2.1.1. Distinct Application and Implementation Languages

First, the general programming method assumes that humans write source programs (or fragments) in an application language (Φ) that is syntactically convenient for humans, and that these programs are then translated into an implementation language (IL) that is more convenient for mechanical interpretation. This prevents the programmer from directly writing ill-formed programs in the common implementation language. Because all programs in the implementation language are generated and manipulated

**Figure 2-1:
General Programming Method**



mechanically, they can be guaranteed to have certain properties: in particular, to be syntactically correct and to be free of possible runtime type errors. (*Runtime type errors* were defined in Section 1.2.1.1.)

This work defines two versions of a simple application language, **Phi**, and a simple implementation language, **IL**. A **Phi Translator**, which translates from Phi to IL, is also defined.

2.1.2. Programs Are Combined

Second, the programming method assumes that useful programs in the common implementation language, possibly from libraries, may be combined to form new programs. In this way, various software components could be reused.

The operation of combining IL programs is not defined here. It is assumed to be handled by whatever particular programming method the programmer uses, and is not essential for discussing the notion of *phases*.¹¹

¹¹Nonetheless, the special data type ERT, described in Section 2.2.1, and the example languages Static-Phi and Static-IL, described in Chapter 4, make it easy to manipulate and combine type-checked program fragments with integrity under program control. In fact, that is precisely the purpose of the unusual (**check-**) constructs of Static-IL listed in section 4.2.3: they take type-checked IL programs (in the form of ERTs), and combine them to produce new type-checked IL programs. A combining program would thus take ERT values as input (from the environment) and produce an ERT value. Section 4.3 discusses the environments required by Static-IL programs and shows examples of ERT values.

2.1.3. Programs Are Instantiated

Finally, the programming method assumes that a program can be specialized, instantiated, refined, or evaluated to form various versions or to compute the final answer. An entire tree of versions might be derived. This aspect is consistent with notions of transformational implementation [Cheatham 81], mechanized top-down stepwise refinement, and partial evaluation [Ershov 77a]. It also means that a version might be generated that would gather program performance statistics, and these statistics could be used in automatically instantiating a more efficient version for those data characteristics [Balzer 83].

Instantiation is defined in this work by the semantics of the Implementation Language (IL), that is, by the IL Machine.

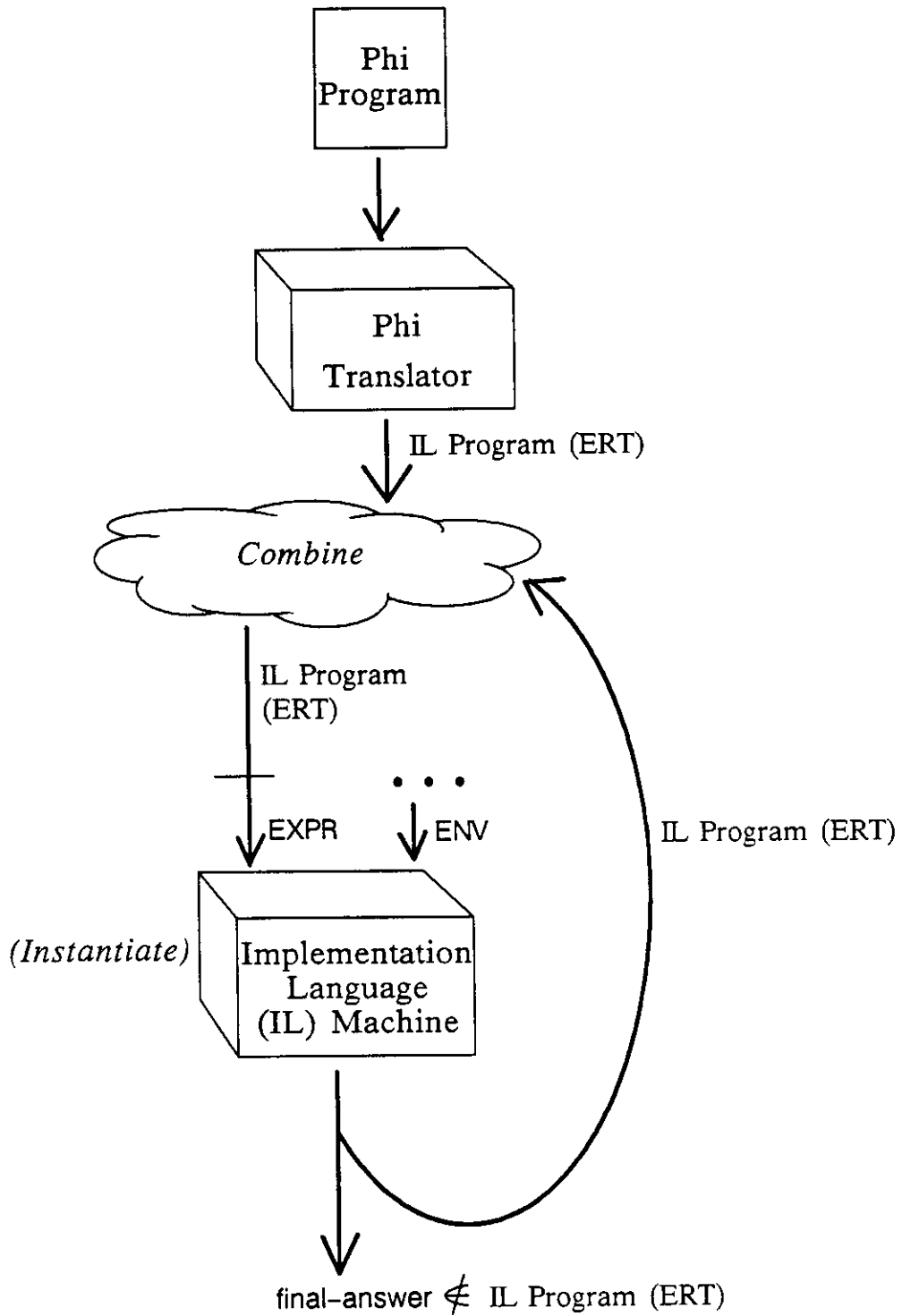
2.2. Specific Programming Method

Before describing the notion of phases, let us first discuss the assumed programming method more specifically as it relates to the succeeding description of phases. Figure 2-2 illustrates the specific programming method. It involves application programs written in Phi, a Phi Translator, IL programs in the form of ERTs (defined below), and an IL Machine.

2.2.1. ERT: Expression, Required-environment, Type

In order to interpret the specific programming method shown in Figure 2-2, we must first define a special data type for representing type-checked program fragments: the data type ERT. An ERT is a triplet having the following components:

Figure 2-2: Specific Programming Method



Expression	An expression in the Implementation Language.
Required-environment	A list of each free variable appearing in the Expression component, paired with its type. Each free variable is listed once, with one type, and no other variables are listed.
Type	The Expression component will evaluate to a value of this type.

The purpose of ERT triplets is to facilitate manipulating programs (expressions), both in the overall programming method and in the implementation language, while ensuring their integrity. We are not interested in just *any* conceivable $\langle e, r, t \rangle$ triplet -- only those that are meaningful, or *valid*, as defined below.¹²

¹²David MacQueen and John Mitchell have aptly pointed out that a valid ERT corresponds closely to the notion of a *typing*. To quote Reynolds [Reynolds 85]:

"Let e be an expression, π (often called a *type assignment*) be a mapping of (at least) the identifiers occurring free in e into types, and σ be a type. Then

$$\pi \vdash e : \sigma$$

is called a *typing*, and read ' e has type σ under π '."

Note that this interpretation is assuming a particular deduction or evaluation mechanism, represented by the symbol " \vdash ". (It might be more precise to subscript this symbol with the name of the deduction mechanism, such as " \vdash_D ".) Similarly, there is a corresponding implied deduction mechanism for ERT triplets, which is given by the semantic rules for interpreting expressions in the Implementation Language.

2.2.2. Valid ERTs

An ERT $\langle e, r, t \rangle$ is **valid** if expression e , evaluated in an environment that *satisfies* the required-environment r , is guaranteed to evaluate to a value of type t . By "an environment that *satisfies* the required-environment" we mean an environment env such that for each variable-type pair $\langle v, t \rangle$ listed in required-environment r , variable v is bound to a value of type t in env .

Every ERT generated by the Phi Translator or the IL Machine is valid.¹³

2.2.3. Interpreting the Specific Programming Method

First, the programmer writes a Phi program.

Next, the programmer invokes the Phi Translator to translate this program to a valid ERT (i.e. an IL program).

The programmer might next use some method of combining various ERTs to create a new ERT.

Then, the programmer creates an appropriate environment, and evaluates the ERT by invoking the IL Machine on this environment and the Expression component of the ERT. The environment supplies the input, and must include values of the proper types for all free variables in the expression. (That is, the environment must *satisfy* the Required-environment component of the ERT, as discussed in Section 2.2.1. The command interpreter used to invoke the IL Machine must enforce this, as discussed in Section 2.2.4.)

¹³To prove this assertion would be quite tedious. The last section of Appendix A includes a brief sketch of how to approach proving it.

The Type component of the ERT tells what type of value the IL Machine will produce, assuming no "compiletime" error occurs. (If such an error does occur, one can either think of the IL Machine as returning some special error value distinct from all other legitimate values, or as returning nothing at all, since evaluation is aborted.) If the Type component is `ert`, the result will be another ERT; otherwise it will be some final answer -- a number, for example.¹⁴ Thus, one knows beforehand whether the result of executing each IL program will be another ERT (another program) or a final answer.

The case when the IL Machine produces another ERT is especially interesting. Since an ERT contains an expression in the Implementation Language, the IL Machine can be viewed as specializing, instantiating, or (possibly) partially evaluating a program, as in the General Programming Method (Section 2.1). But it can also be viewed as *compiling* a program, though the source and object languages are the same. This is further explained in Chapter 3.

Note that it is trivial to determine whether the result produced by the IL Machine will be a final answer: the Type component of an ERT specifies the type of value that will be produced then the Expression component is evaluated. Thus, if the Type component is anything other than the literal `ert`, the result of the phase evaluating the expression will be a final answer. This is evident in the examples of Section 4.4.

¹⁴A final answer is defined as any value other than an ERT, that is, it is not another program. It might be a number, boolean, string, or other such basic value. Conceptually, a final answer might be an entire file, though in our simple pedagogical languages it will not.

2.2.4. A Command Interpreter

Certain aspects of the programming method shown in Figure 2-2 must be done by the human. For example, the human must write the original Phi program, invoke the Translator on it, combine program fragments (ERTs) as desired, supply the desired environment, and invoke the IL machine on the Expression component of the desired ERT. The simplest method of doing these things is to provide a command interpreter -- most naturally written in the Phi language itself -- and this is what we will assume, though any other more automated method is possible as well. It is the command interpreter's responsibility to ensure that the expression and environment actually given to the IL Machine are syntactically correct, type correct, and compatible. However, the use of ERTs makes this very easy to enforce, especially since every ERT produced by the Phi Translator or the IL Machine is guaranteed to be syntactically correct and type correct, and the Required-environment explicitly lists the identifiers and types of values required in the environment.

2.2.5. Environments

An environment simply provides bindings of identifiers to values. As shown in Figure 2-2, along with each IL program (EXPR), the IL Machine must be given an environment (ENV) that supplies values of the correct type for all free variables in the IL program (EXPR). In our simple model, an IL program's input must be supplied via the environment; that is, the IL program might have a free variable representing the program's input, and the environment would have to supply a value for that free variable, thus providing the program's input.

For example, consider the following trivial program that computes the *cosine* of a number, x .

(cos x)

The free variable x represents the program's input, and the free variable *cos* refers to a standard *cosine* trigonometric function. Thus, the environment for this program must be constructed to include bindings for x (a number), and *cos* (a function from numbers to numbers). Typically, the binding for *cos* would come from a standard library, whereas the binding for x would be explicitly provided by the user.

We do not show how environments are generated, but the command interpreter can provide ways of creating, combining, and storing environments, while keeping track of the types of the variables defined in them. Pebble [Burstall 84], for example, uses bindings as first-class values, and provides operations for creating and combining them.

Section 4.3 explains more about the environments required for Static-IL programs. (Static-IL is discussed in Section 4.2.)

2.3. Motivating Example: General Purpose Sorting Function

This section describes a hypothetical example of how general-purpose reusable programs might be created and used. The purpose of this example is to provide a tangible goal to guide the reader's intuition through the rest of this work, where the notion of *phases* is explained. The reader may wish to skip this section at first, and return to it later as needed.

Bear in mind that the languages discussed in this work are provided for

pedagogical purposes only. They would not be practical for real-life applications such as the motivating example described in this section. However, these pedagogical languages should demonstrate the basic semantic notions necessary in a full, usable language that could be practically applied to the example below.

2.3.1. The Desire for a General-Purpose Sorting Function

Consider the problem of providing a truly general-purpose sorting function. Such a function should be able to efficiently handle a wide range of sorting needs, from sorting a small fixed number of items in the computer's primary memory, to sorting thousands of records in primary memory, to sorting millions of records in secondary memory such as disk or tape.¹⁵

Clearly, it is impossible for a single sorting function to fill all of these needs efficiently enough to be generally useful, because there are many different algorithms that are appropriate for different needs. Any single program that tried to meet all needs would be much too large to be practical for the smaller cases.

¹⁵This example comes from another author, but we have been unable to determine whom.

2.3.2. A Sorting Function Generator

But consider a sorting function *generator*. This generator could be given input characterizing a particular sorting need, and would produce a sorting function custom-tailored for that application. Input to the generator might include parameters describing the data types to be sorted, where the data are stored, the type of algorithm to be used, or even a characterization of the generated program's expected input data distribution. The generator would use this information to choose the most appropriate algorithm (from some repertoire) and produce the most efficient data structure declarations.

An automatically generated sorting function probably would not be quite as efficient in every case as a sorting function that a programmer could write from scratch. However, it could be good enough in most cases that it would be far more cost-effective to use the automatically generated version than to write a new one. This is a fundamental assumption behind the desire for reusable software.

2.3.3. Explicit Generation vs. Partial Evaluation

The sorting function generator could be written in two ways: it could explicitly manipulate program fragments for the generated program, or it could be written as one big parameterized sorting program that is partially evaluated to produce a small specialized version. Ignoring the lack of strong typing in LISP, the approach of explicit manipulation might correspond to LISP programs that construct other LISP programs as S-expressions. In the partial evaluation approach, the language would have to allow types as first-class values so that data type declarations could be parameterized by input values, and the partial evaluator would manipulate program fragments to

produce a specialized version -- the program would not express this manipulation explicitly. Regardless of which approach is taken, the important point here is that the work of *producing* the specialized or generated sorting program must be separated from the sorting program's *execution*. For this discussion, we will assume that explicit generation is used.

2.3.4. Phases Used

Let us now clearly distinguish between the act of executing the sorting program generator and the act of executing the generated sorting program. These executions correspond to two acts of *instantiation*, shown in Figures 2-1 and 2-2, or two *phases*, n and $n+1$, as described in Chapter 3.

To ensure strong typing, both the sorting program generator and the generated program must be guaranteed against runtime type errors. In the *phase* parlance of Chapter 3, if the generator is to be executed in phase n , it can be type checked in phase $n-1$; if the generated program is to be executed in phase $n+1$, it can be type checked in phase n .

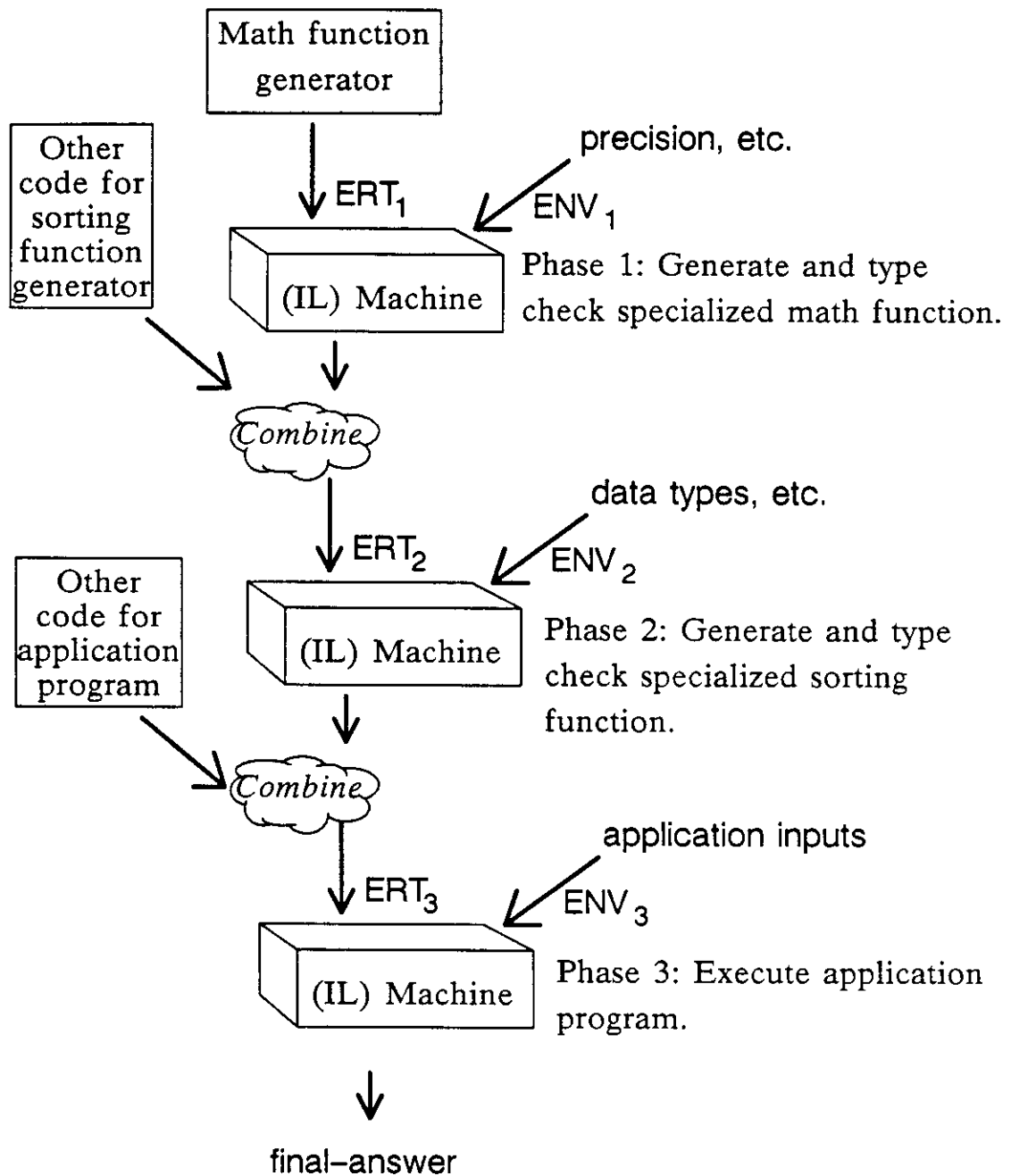
2.3.5. Generalizing Further

So far we have focused on the application program's need to use a general-purpose function. To generalize the example further, suppose that the sorting program generator also uses some general-purpose mathematical function that also must be specialized before being used. Thus, a math function generator would produce a specialized version of the math function, which would be used in the sorting function generator to produce a specialized sorting function, which would be used in some application

program. Again, in the parlance of Chapter 3, the math function generator would be executed in phase $n-1$ to produce and type check the specialized math function, which would be used by the sorting function generator in phase n . These phases are illustrated in Figure 2-3.

In summary, general-purpose function generators can be used to produce specialized functions, which may themselves be used by other general-purpose function generators.

Figure 2-3: Phases of Sorting Example



Chapter 3

The Conceptual Model of Multiple Phases

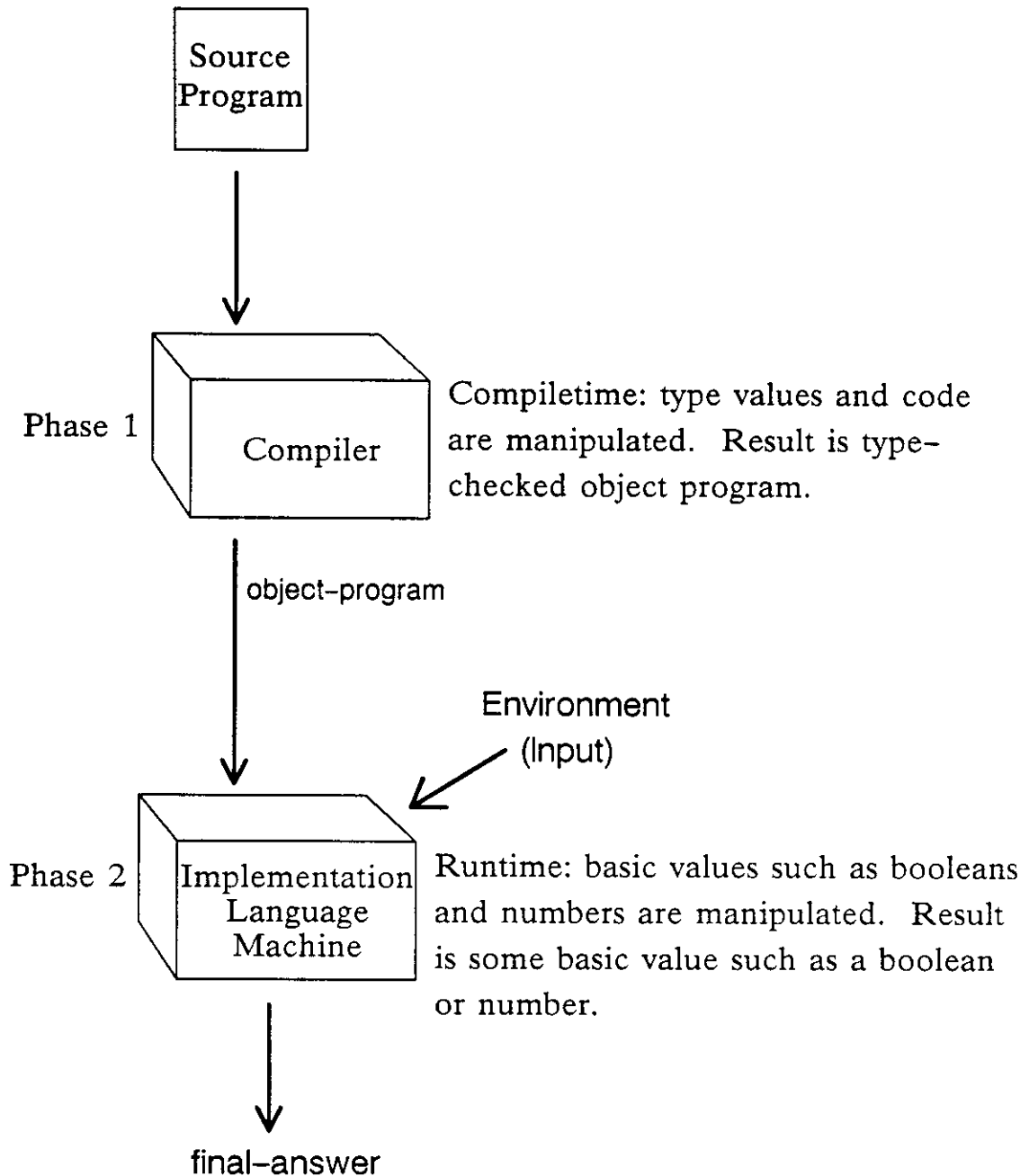
This chapter describes the conceptual model of multiple strongly typed evaluation phases. Proper understanding of the conceptual model is critical in understanding the Phi language, the Phi translator, and the Implementation Language.

3.1. Arriving at Phases by Extending Compiletime

The conceptual model of phases is best understood by presenting the arguments that led to its development. We begin with a simple conceptual view of traditional compiletime and runtime, shown in Figure 3-1.

First, a source program, written in a strongly typed language, is compiled into an object program. This step is Phase 1 -- compiletime. During this phase, the compiler manipulates type values and program code, and as a result produces an intermediate *object* program that is guaranteed free of

**Figure 3-1:
Traditional Compiletime and Runtime**



runtime type errors.¹⁶

The object program, with a suitable environment, is then executed on an implementation machine. This step is Phase 2 -- runtime. During this phase, basic values such as numbers, character strings, and booleans are manipulated, and the result of the computation is some basic final value such as a number, a character string, a boolean, or, conceptually, a file. The environment defines all identifiers that are not locally declared in the program, that is, it provides bindings for all of the program's free variables.

3.1.1. Generalizing Compiletime

Let us now view the compiler as *executing* the source program to produce the object program, and allow the programmer to express types as first-class values that are manipulated during compiletime. And to provide really useful expressive power, let us also allow the programmer to express other types of values, for example, numbers and booleans, at compiletime, and to write arbitrary compiletime expressions and functions involving these values.

Now, with values of various kinds (numbers, booleans, and of course

¹⁶The question of whether there could be type errors in a program's input sometimes arises here. For example, an input operation requiring a number could instead be given some meaningless character string. This problem can be avoided by only providing an input operation that always reads characters, and forcing type conversion to be accomplished by ordinary functions under programmer control. Thus, for example, the input sequence "123" would be read as the *characters* "1", "2", "3" of known type, and then converted by the program to the *numeric* value 123.

For simplicity, the simple pedagogical languages described in this work do not include input or output operations.

types) being manipulated at compiletime, it is conceivable that a so-called "runtime" type error could occur during compiletime. For example, one may mistakenly try to add a number to a type during compiletime. Therefore, we add another phase -- a pre-compiletime phase -- that does the type checking required to ensure that no "runtime" type errors can occur during compiletime.

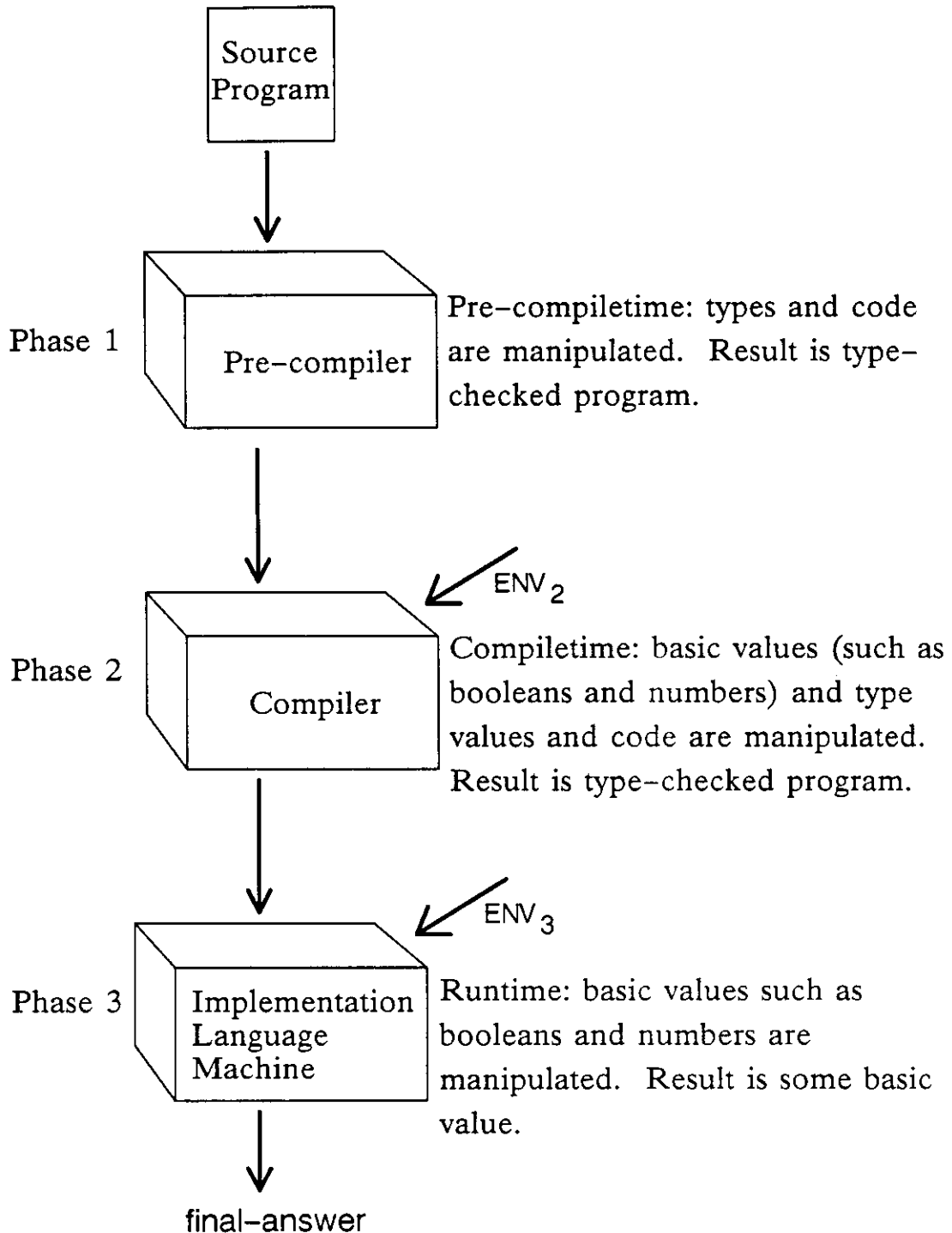
Our conceptual model, at this point, is shown in Figure 3-2. Phase 1, the pre-compiletime phase, now manipulates type values and program code, and as a result produces a program that is guaranteed not to commit a "runtime" type error when executed during the next phase. Phase 2, compiletime, now manipulates numbers, booleans, type values, and program code, and produces a program that is guaranteed free of runtime type errors. Phase 3, runtime, manipulates numbers and booleans as before, producing a final answer (number, boolean, etc.).

3.1.2. Generalizing Pre-compiletime, And So On . . .

At this point, we can make two observations. First, the pre-compiletime phase is now performing a role completely analogous to the role compiletime had played. Hence, we can apply the same reasoning to generalize pre-compiletime, and add a pre-pre-compiletime phase, and so on, thus potentially allowing an unbounded number of phases. Each phase except the last produces a type-checked program for the next phase.

Second, we observe that the operations performed by the compiler have now become a superset of the operations performed at runtime. Hence we can unify the two so that one Implementation Language (IL) Machine fills both roles. The resulting conceptual model is described in Section 3.2.

Figure 3-2: Pre-Compiletime Phase Added



3.2. Conceptual Model of Multiple Phases

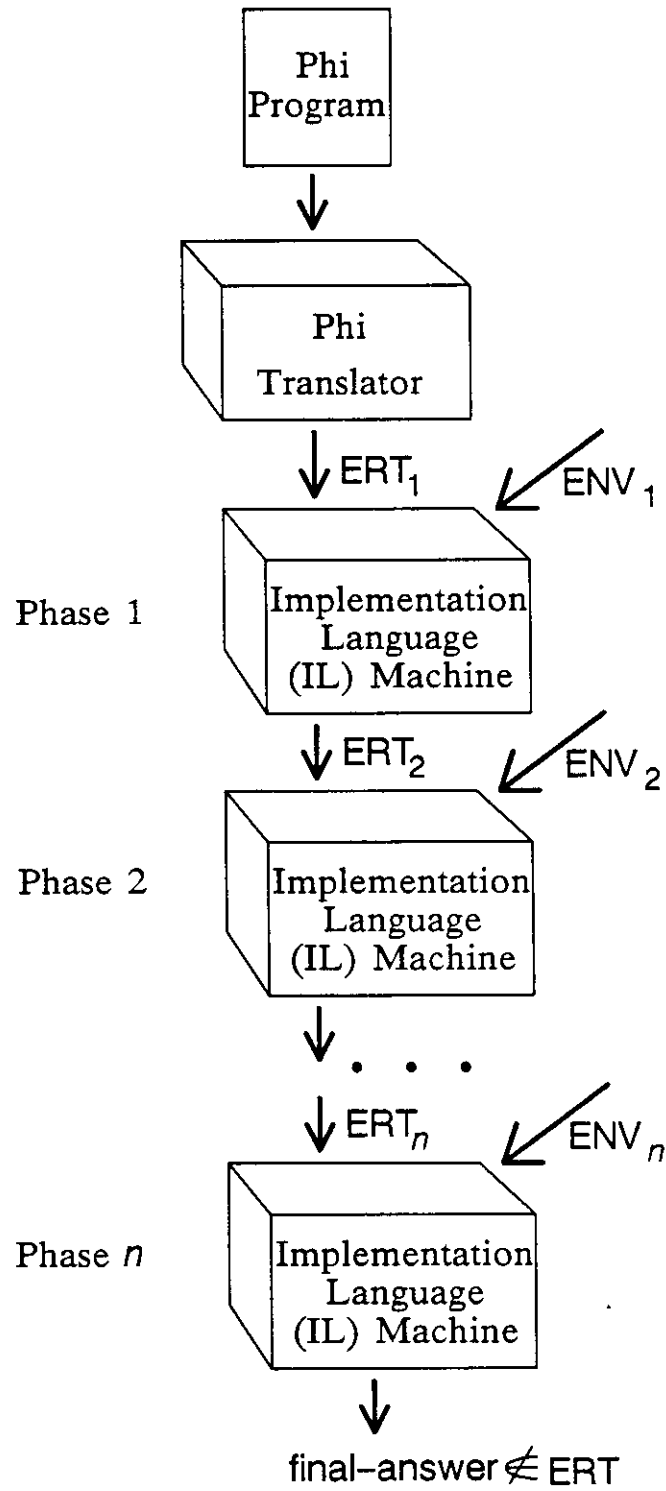
Figure 3-3 illustrates how a Phi program is translated and then, in effect, executed through several intermediate *phases* before producing a final result.

Note the correspondence between Figure 3-3 and the specific programming method illustrated in Figure 2-2. The loop shown in Figure 2-2 is unfolded in Figure 3-3; thus the conceptual model shows several repetitions of the IL Machine -- one for each time it is invoked. Also, for simplicity, the *combine* action in Figure 2-2 is not shown in Figure 3-3.

3.3. Interpreting the Conceptual Model

A Phi program is first translated to ERT_1 . The Expression component of ERT_1 is then executed on the IL Machine in a suitable environment ENV_1 -- this is phase 1 -- to produce ERT_2 . The Type component of ERT_1 specifies the type of value that will be produced by phase 1. For the first phase, it is always *ert*, indicating that another ERT will be produced. Similarly, the Expression component of ERT_2 is then executed in phase 2 to produce ERT_3 , and so on. The Type component of ERT_2 specifies the type of value that will be produced as a result of phase 2, etc. The result of some phase n is considered the final result of the computation because it is not an ERT. That is, the Type component of ERT_n indicated that the result would be something other than another ERT. Thus, the original Phi program could be viewed as a meta-program because, in effect, it denotes a series of programs ERT_1, \dots, ERT_n .

Figure 3-3: Conceptual Model of Multiple Phases



3.3.1. Properties of the Conceptual Model

The conceptual model has the following important properties:

- Each phase does the type checking necessary to ensure that no runtime type errors are possible during the next phase.
- No runtime type errors are possible during the first phase, either.
- Every ERT produced by the Phi Translator or the IL Machine is valid.¹⁷ Specifically, the Expression component is guaranteed syntactically correct and type correct.
- The type of each subexpression is computed at least one phase before the value of that subexpression is computed. Similarly, the type of the program's result is known before the program is phase evaluated (i.e. it is given as the Type component of an ERT).
- Each phase acts as compiletime for the next phase, and as runtime for the previous phase. Thus, the terms "compiletime" and "runtime" are relative. These terms will still be used in the rest of this work -- they are still meaningful terms -- but the reader should recognize that their meanings are relative to other implied runtime or compiletime phases.
- The Phi Translator does no type checking -- it will produce a valid ERT, free of possible runtime type errors, for any syntactically legal Phi program. This is explained in Section 3.3.3.

¹⁷As defined in Section 2.2.2.

3.3.2. Resolving the Conflict Between "Strong Typing" and "Types as First-Class Values"

Section 1.2.3.3 points out the inherent conflict between strong typing and the desire for types as first-class values. In our model of multiple phases, types are indeed allowed as first-class values, yet every phase is strongly typed. How is the conflict avoided in our model?

In general, types manipulated as first-class values during one phase become invariants of the next phase, in the sense that a type used in a declaration represents an invariant. If an identifier is declared to be some type, that type represents an invariant on the kinds of value that may be bound to that identifier. Similarly, if a function's return value is declared to be a certain type, that type represents an invariant on the kinds of value that the function may return.

In our model it is not possible to use a type, computed as a first-class value, as an invariant of the same phase during which it was computed. Type values computed in one phase have no bearing on the types of the expressions executed during that same phase.¹⁸ One can compute an arbitrary type value during one phase, but that type value can only be used in declarations pertaining to subsequent phases -- not in the declarations

¹⁸This property is readily evident in Static-IL, presented in Section 4.2. Expressions in Static-IL are type checked and generated in the form of ERTs, and in doing so, types are computed as first-class values. However, there is no construct in Static-IL for *evaluating* an ERT. That is, there is no provision for invoking the Static-IL Machine from within Static-IL. Hence, there is no way for the type values, computed in one phase, to have any effect on the types of the identifiers or expressions evaluated during that same phase.

pertaining to that same phase.¹⁹ For example, in the same phase, one cannot both compute the type used to declare an identifier, *and* bind a value of that type to the identifier. The type of the identifier must be computed during at least one phase *before* the identifier may be bound to a value of that type. Thus, the notion of separate phases prevents any possible circular dependency between an object's type and its value.

3.3.3. The Paradox of Strong Typing Without Prior Type Checking

We mentioned that every phase is strongly typed, and that the IL program for every phase -- except the first -- is type checked by the previous phase. We require that the first phase also be strongly typed, yet we also mentioned that the Phi Translator does *no* type checking. How can we ensure that the IL program produced by the Phi translator does not contain any runtime type errors if the IL Translator does no type checking? The answer is simple: the translator produces an IL program in which every subexpression evaluates to a value of the same type: type ERT.

This means that the only operations performed during the first phase are manipulations of program fragments. This makes sense when one considers: what if it *weren't* true. That is, suppose some other operation -- addition of two numbers, say -- could be performed during the first phase. Then, to guarantee that this operation could not involve a runtime type error, there would either have to be another previous phase or the translator would have to do some type checking.

¹⁹Conceivably, the type may even be computed by a recursive function, as mentioned in Section 6.3, though for simplicity recursive functions are not provided in the Static-Phi and Static-IL languages described in Chapter 4.

Hence, every variable is type ERT initially, and every IL program produced by the translator evaluates to an ERT (assuming no compiletime errors occur during evaluation). (If a compiletime error does occur during evaluation, the program can either be thought of as returning some special error value, distinct from all other values, or as returning nothing, since the evaluation is aborted.)

3.3.4. All Expressions Start Out Type ERT

If we view the IL programs $ERT_1 \dots ERT_n$ in Figure 3-3 as representing successive versions of the initial Phi program, then the type of every subexpression or variable in the initial Phi program starts out as ERT, and remains ERT until some phase when it becomes fixed as some basic type, such as a number or a boolean (any type other than ERT). Finally, during the following phase, the expression or variable will have a value of that type (number or boolean).

This one-way progression represents the accumulation of information about the expression or variable. Type ERT means that nothing is known about the expression or variable. Then, during some phase, the type of the expression or variable is known (number or boolean, for example). Finally, during the next phase, the specific value of the expression or variable is computed. This subject is mentioned further in Section 7.2.8.

3.4. Assigning Computations to Phases

Given a source program, we need some way to decide during what phases its various subcomputations should be performed. For example, we require that the type of a function's formal parameter be computed at least one phase before the function can be applied to any actual arguments. There are two basic approaches we can take; the first of these has two variations.

1. **Static determination.** The phase for each computation is fixed during translation, before the first phase. This approach most closely follows the reasoning presented in Section 3.1, which led to the idea of multiple strongly typed evaluation phases, and this is the approach on which this work was initially based. There are two sub-options possible under this approach:
 - a. The source program can *explicitly* indicate which computations are to be performed during each phase. This was the original approach conceived as "multiple strongly typed evaluation phases", and is described in Section 4.
 - b. The Phi Translator might *infer* which computations should be performed during each phase. This approach was not pursued in this work. We do not know how difficult this alternative might be, or what problems it might present. It is open for future research, as mentioned in Section 7.2.9.
2. **Dynamic determination.** The phase for each computation is determined during the various execution phases, and depends on the environments supplied during the previous phases. This would allow phases to achieve the effect of partial evaluation, because the types and values of different free variables could be "fixed" as desired during different phases. The essential distinctions between this kind of phase evaluation and partial evaluation are that, under phases, the same machine would be used to perform "partial" and "full" evaluation, there is a rigid requirement of strong typing in each phase, and the type of result -- either the final answer or another program -- would be known in advance.

This approach has not been fully explored, but the possibility is discussed in Chapter 5.

3.5. How Many Phases Are Required?

How many phases will be required to execute a given Phi program to a final answer? In general, the answer depends on the program, whether a model of static or dynamic determination of phases is used, and might depend on the environments provided in the various phases.

Any given program will always require some *minimum* number of phases before it can produce a final answer. For the Static-Phi language described in Section 4.1, an algorithm (*Count*, defined in Appendix A) is used to compute this minimum based on the lexical nesting level of *emits* and *evals* in the original Static-Phi program, and, hence, it cannot be infinite for a finite-sized program.²⁰ For example, the program demonstrated in Section 4.4.3 requires three phases, whereas the program in Section 4.4.9 requires four phases.

What about using more phases than the minimum? When phases are determined statically, there is little flexibility for extra phases, because a given program would expect certain inputs, via the environments, in certain phases. (Section 4.3 discusses environments for Static-Phi.)

If phases were determined dynamically, with each phase performing the function of partial evaluation, then extra phases might freely be used.

²⁰We do not know if there might be any other reasonable language in which a well-formed program could require an infinite minimum number of phases. We suspect not, and the question is not considered here.

Partial evaluation is defined to preserve the semantics of the original program, so extra phases should certainly cause no harm, and they may improve the efficiency of later phases by allowing the values of some expressions to be pre-computed. Of course, if there are no more expressions that can be pre-computed, adding an extra phase does nothing useful. As a trivial example, consider the program consisting only of the variable x . If no final value is given for x , x will just partially evaluate to itself. That is, the program will be partially evaluated perfectly well, but no useful work will be done because no further reduction is possible until a final value is supplied for x .

Chapter 4

Static Determination of Phases

This chapter informally describes the originally conceived system of multiple phases, in which the phase for a particular subcomputation is explicitly denoted in the source program. That is, the phase for a given subcomputation is determined statically during translation. We demonstrate this approach by defining a source language, *Static-Phi*, a *Static-Phi Translator*, and an implementation language, *Static-IL*. More precise semantic definitions are given in Appendix A.

The reader well-versed in the typed lambda calculus may wish to skim Section 4.1, which describes *Static-Phi*, noting the special **emit** and **eval** constructs, and then turn directly to Section 4.2, which describes *Static-IL*. Section 4.2.3 is important because it discusses the unusual language constructs in *Static-IL*. Finally, the reader is strongly urged to read the discussion of the two examples in Sections 4.4.3 and 4.4.9 to gain an appreciation of how phases work.

4.1. The *Static-Phi* Language

The *Static-Phi* language is expression oriented, and looks like a simple typed lambda calculus [Barendregt 84] with two extra constructs added. Types are unrestricted first-class values; wherever a type is required, any arbitrary expression that evaluates to a type may be given. There is no

modifiable store, or assignment operation. There is one abstraction operator, λ , for data abstraction, function abstraction, type abstraction, and code (ERT) abstraction.

4.1.1. Conventional Static-Phi Language Constructs

The Static-Phi language includes the following basic forms:

- constant* A literal constant, for example, a number **1234**, a truth value **false**, or a type constant **number**, **bool**, **ert**, or **type**. Type constant **type** refers to the type of types; **ert** is the type of ERTs, described in Section 2.2.1.
- id* An identifier (variable). An identifier always evaluates to the value bound to it in the environment.
- $\lambda id : expr_d \rightarrow expr_r . expr_{body}$ For creating an unnamed function abstraction. $Expr_d$ and $expr_r$ are arbitrary expressions that must evaluate to types; they declare the types of the domain and range of the function, that is, $expr_d$ is the type of the formal parameter *id*, and $expr_r$ is the type of the function's return value. $Expr_{body}$ is the body of the function. Because we require "compiletime" type checking (that is, one phase before "runtime"), the formal parameter type will be evaluated one phase before the function value (closure) is created. That is, if the function is to be applied in phase *i*, the type of the formal parameter will be computed in phase *i-1*.
- $(expr_f \ expr_x)$ Function application. $Expr_f$ is an arbitrary expression that must evaluate to a function; $expr_x$ will evaluate to the actual argument. The type of the actual argument must match the declared type of the formal parameter for the function; this is checked during the phase before the function is applied. The function application always occurs during the same phase that the actual function

value is created, regardless of any nesting inside `emits` or `evals` (described below).

(`funtype` $expr_d$ $expr_r$)

Standard function for constructing the types of functions. The subexpressions are evaluated (they evaluate to types) and paired to represent the types of the domain and range of a function. $Expr_d$ is the domain type; $expr_r$ is the range type. Of course, both subexpressions must be type **type**; this is checked one phase before the function type is to be constructed and returned.

4.1.2. Normal Runtime Phase

Normal runtime phase refers to the phase in which a particular operation is actually performed (as opposed, say, to the phase in which the operation is type checked). Within a single program the normal runtime phase will be different for different instances of different operations. For example, the type expression for a function's formal parameter might use an operation that is also used in the body of the function. Used in the formal parameter type expression, the operation's normal runtime phase will be one phase sooner than for the instance of the operation that appears in the body of the function. "Normal runtime phase" is usually used as a comparative term, to contrast the different phases when two operations are performed.

By altering the normal runtime phase of an operation, one can cause the operation to be performed during some phase earlier or later than it would otherwise be performed. Basically, if an operation is used to compute a type that will be used to type check a subsequent phase, then one would want the normal runtime phase of the operation to be one phase earlier than it otherwise would be. Or, if an operation is used to explicitly generate some

code (an ERT) that is to be executed in a later phase (as with macro expansion), then one would also want the normal runtime phase of the operation to be one phase earlier than it otherwise would be. On the other hand, if the operation in question were a part of the generated code, one would want its normal runtime phase to be one phase later: that is, the normal runtime phase of the operations that are doing the generation should be one phase earlier than the normal runtime phase of the operations in the generated code.

The normal runtime phase of a construct is altered in three ways: by being inside an `emit` (discussed below), by being inside an `eval` (also below), or by being in a function abstraction's range or domain type expression. The normal runtime phase for a function abstraction's range or domain type expression is implicitly one phase earlier than the normal runtime phase for the function, since the function must be type checked during the phase before it is applied. `Emit` and `eval` are used to explicitly change the normal runtime phase of an expression: `eval` makes the normal runtime phase one phase earlier, while `emit` makes it one phase later. These are discussed below in Section 4.1.3, and are more precisely defined in the formal semantics given in Appendix A.

4.1.3. Some Unusual Constructs

In addition to the familiar constructs outlined in Section 4.1.1, Static-Phi also includes the following unusual forms.

(`eval expr`) The normal runtime phase of *expr* is one phase earlier than in the surrounding context. Note that the domain and range type expressions in the λ construct, $expr_d$ and $expr_r$, are effectively inside an implicit `eval`, because the

types need to be computed one phase before the function value (closure) is created.

(emit *expr*) The normal runtime phase of *expr* is one phase later than in the surrounding context.

Note that our **eval** is very different from the LISP EVAL. Our **emit** and **eval** forms are only used during translation. They are not executable notions, and there are no Static-IL syntactic forms that correspond to them.

Note also that **emit** and **eval** cancel each other out, in a manner analogous to the LISP back-quote (``) and comma (",") macro constructs. Thus, **(emit (eval *expr*))**, **(eval (emit *expr*))**, and *expr* are entirely equivalent in Static-Phi.

4.1.4. Examples

This section shows some simple examples of Static-Phi programs. Section 4.4 shows how each of these examples would be translated to Static-IL programs and appear in various phases. The explanations of the identity function examples in Sections 4.4.3 and 4.4.9 give the flavor of what the various phases do. We begin here with trivial examples and work up to more interesting cases.

4.1.4.1. F Twice

(f (f x))

Some function *f* is applied twice to an argument *x*.

4.1.4.2. Identity Abstraction

$$\lambda x : \text{number} \rightarrow \text{number} . x$$

An unnamed identity function that takes a number and returns that same number.

4.1.4.3. Identity Application

$$(\lambda x : \text{number} \rightarrow \text{number} . x \ 5)$$

The identity function from the previous example is applied to the number 5. The final result will be 5.

4.1.4.4. Function Abstraction

$$\lambda x : \text{number} \rightarrow \text{number} . (\text{succ} (\text{succ } x))$$

If *succ* is the successor function on numbers, defined in the environment, this is an unnamed function that adds 2 to its argument.

4.1.4.5. Function Application

$$(\lambda x : \text{number} \rightarrow \text{number} . (\text{succ} (\text{succ } x)) \ 5)$$

The function from the previous example is applied to 5. The final result will be 7.

4.1.4.6. Higher Order Function Abstraction

$$\lambda f : (\text{funtype number number}) \rightarrow \text{number} . (f (f x))$$

This function takes another function as an argument and applies it twice to some free variable *x*. The actual parameter must be a function from numbers to numbers.

4.1.4.7. Higher Order Function Application

$(\lambda f : (\text{funtype number number}) \rightarrow \text{number} . (f (f x)) \quad g)$

The higher order function from the previous example is applied to g , which must be a function from numbers to numbers. Thus, function g is applied twice to the free variable x . The program is equivalent to:

$(g (g x))$

4.1.4.8. Identity-Function Type Abstraction

$\lambda t : \text{type} \rightarrow \text{ert} . (\text{emit } \lambda x : t \rightarrow t . x)$

This function takes a type t and returns code (an ERT) that will become an identity function in the next phase. The generated identity function will be specialized for type t , and may only be applied to values of type t .

There are two function abstractions in this example: the outer function abstracts the type variable t in one phase, and the inner function abstracts the variable x in the next phase. Note the `emit` surrounding the inner function abstraction. The `emit` informs the Static-Phi Translator that the inner function abstraction is to be created one phase later than the outer function abstraction. This is required because the outer function abstraction is manipulating a type value that will be used in type checking the inner function. Hence, during the phase when the outer function is created and applied, the inner function is just treated as code (an ERT), and is type checked. The outer function is acting like a macro in returning the code (an ERT) instead of returning a function value (or closure).

The outer function cannot *both* compute the type t as a first-class value *and* return the inner function as a function value (closure) during the same phase, because, to enforce strong typing, the inner function must be type

checked during the phase *before* it is used as a function value. Therefore, if the `emit` were omitted, a compiletime error would occur when the inner function was being type checked.

4.1.4.9. Identity-Function Type Application

`(λ t : type → ert . (emit λ x : t → t . x) number)`

The identity-function generator of the previous example is applied to type `number` to generate an identity function from `numbers` to `numbers`.

4.1.4.10. General Type Abstraction

`λ t : type → ert . (emit λ f : (funtype t t) → t . (f (f x)))`

This function takes a type `t` and returns code (an ERT) that will become a function in the next phase. The generated function will take any function from `t` to `t` and apply it twice to the free variable `x`.

This example demonstrates how types may be manipulated as first-class values during one phase, yet become invariants of the next phase. The outer `λ` creates a function that takes (and could manipulate) a type as a first-class value. However, it returns code (an ERT) that has been type checked using this type. This returned code happens to be the code for a function abstraction. (Incidentally, free variable `x` is also type checked when the ERT for the function is generated and type checked.) Section 4.4.10 shows how this example would appear in various phases.

4.1.4.11. General Type Application

```
(λ t : type → ert . (emit λ f : (funtype t t) → t . (f (f x))) number)
```

The ERT-returning function of the previous example is applied to type `number`.

4.1.4.12. Macro Abstraction

```
λ m : ert → ert . (m (m x))
```

This function takes some code `m` (an ERT) and returns code that applies `m` twice to some free variable `x`.

Note that the formal parameter `m` and the function's return type are both `ert`, indicating that this function will take code (an ERT) as its argument and return code (an ERT) as its result. This function manipulates code, much like a macro.

4.1.4.13. Macro Application

```
( λ m : ert → ert . (m (m x))  
  (emit λ y : number → number . (succ (succ y)))  
 )
```

The macro of the previous example is applied to code which will become a function to add 2 to its argument. Note that the actual argument is surrounded by an `emit` so that the (macro) function of the previous example will operate on it as code (an ERT) rather than as a function value (or closure). Thus, the outer function treats the inner function as code during one phase, and the inner function becomes a function value (closure) during the next phase. If the `emit` were omitted, the outer function could operate on the inner function only as a function value (closure), not as code. Section 4.4.13 shows the IL code that results from translating and executing this example through the necessary phases.

4.2. The Static-IL Language

As shown in the conceptual model (Figure 3-3), a Phi program is not executed directly, but is first translated into a corresponding Static-IL program. The translator is defined in Appendix A, though examples of translation are given in Section 4.4. This section describes the Static-IL language, which includes some unusual language constructs for creating and combining type-checked program fragments in the form of ERTs.

Syntactically, Static-IL looks like an untyped lambda calculus. In fact Static-IL is typed, though type declarations are not explicit. Under the programming method shown in Figure 2-2, the Static-IL Machine is given only Static-IL expressions that are guaranteed free of runtime type errors or unbound variables, and it generates Static-IL expressions only within valid ERTs.²¹ Since a valid ERT triplet includes a list of all the Static-IL expression's free variables and their types, and the type of the expression, Static-IL expressions should be regarded as typed.²² We speak of Static-IL expressions as being well typed in the same sense that one would speak of the object code for a compiled Pascal program as being well typed, even though the type information from the source program is stripped out after being checked, when the object code is generated.

²¹"Valid ERT" was defined in Section 2.2.2.

²²John Mitchell and David MacQueen have pointed out that it may be better to regard the implementation language as consisting of the entire ERT triplet (rather than just the Expression component), since the R (Required-environment) and T (Type) components of the ERT triplet contain the type information for the E (Expression) component.

4.2.1. Lexical Scoping, ERTs, and Macros

Static-IL expressions are lexically scoped. Nonetheless, if ERTs are explicitly manipulated by the programmer, just as with conventional macros, it is possible to generate new expressions in which free variables have become "captured" by local declarations. Note that this is possible only in program fragments (ERTs) that are explicitly being *constructed*, as first-class data objects. When an expression is *executed*, that expression is absolutely lexically (or statically) scoped, and no such anomalies are possible.

The examples below illustrate how, in constructing an expression by manipulating ERTs as first-class values, a variable can appear to become "captured", as with macros. In the following Static-Phi program, *x* will evaluate to the ERT representing the outer *z*, thus causing the outer *z* to be placed into the scope of the inner *z*.

```
λ z : number → (funtype number number) .
  (eval
    (λ x : ert → ert .
      (emit λ z : number → number . x)
      z
    )
  )
```

For example, the following Static-Phi program (which simply supplies actual parameters for the functions in the preceding program),

```
(
  (λ z : number → (funtype number number) .
    (eval
      (λ x : ert → ert .
        (emit λ z : number → number . x)
          z
        )
      )
    )
  5
)
10
)
```

will be phase evaluated to produce the following Static-IL program,

```
(apply (apply (lambda z (lambda z z)) (quote 5)) (quote 10))
```

which evaluates to 10.

The behavior illustrated above is quite intentional -- it was not an oversight -- though it is different than one might naively expect. The explicit intent here is to manipulate program fragments (ERTs) to construct new programs with new semantics. This behavior is useful for program-writing programs, and is analogous to the behavior of conventional macros. Also, bear in mind that under no circumstances can this behavior cause a runtime type error. Any attempt to cause a type mismatch in the constructed code will be detected as a compiletime error, one phase before the constructed code can be executed.

4.2.2. Conventional Lambda Calculus Operations

The following Static-IL primitives look and function exactly like the basic operations of an untyped lambda calculus, written in the style of LISP:

ev Any quoted expressible value. The value *ev* is simply returned, unevaluated. In Static-IL, constants appear as explicitly quoted values.

id An identifier. Its value is simply retrieved from the environment.

(*lambda id expr*) Function abstraction. *Id* is the formal parameter, *expr* is the function body. A lambda abstraction evaluates to a closure, consisting of the current environment, the formal parameter, and the function body.

(*apply expr_f expr_x*) Function application. *Expr_f* evaluates to a function closure; *expr_x* is evaluated and becomes the actual argument. The function application has already been type checked during the previous phase.

(*funtype expr_d expr_r*) This operation is used to generate the type of a function. Subexpressions *expr_d* and *expr_r* are simply evaluated in the current environment; they evaluate to types. These types, *type_d* and *type_r*, are used as the domain and range types of the function type that is returned.

The returned function type is represented as a pair, tagged with the word *fun*: $\langle \text{fun } type_d, type_r \rangle$. For example, $\langle \text{fun number number} \rangle$ represents the type of a function that takes a number and returns a number.

(*incr expr*) Increment. This operation returns the value of the expression plus one. There is no corresponding operation in Static-Phi: *incr* is only included in Static-IL to make

the examples in Sections 4.1.4 and 4.4 more interesting. (In the examples of Section 4.4, `incr` is used to implement the `succ` function, which is assumed to be supplied in the environment.)

These operations are not discussed further here.

4.2.3. Some Unusual Operations

The purpose of the conventional Static-IL operations listed above is to do conventional computations -- to manipulate basic values as in a lambda calculus. The only perceptible difference is that types are also manipulated along with other basic values.

In contrast, the rest of Static-IL's primitive operations do not look so conventional. Their ultimate purpose is to produce type-checked program fragments (ERTs). That is, the ultimate purpose of the operations listed below is to type check and generate the conventional operations listed above. All of the Static-IL constructs discussed below return ERTs as their result. Several of them involve a parameter n , which is a constant, determined during translation, that indicates how many phases to wait before generating one of the conventional operations. The Static-Phi translator uses the `emits` and `evals` to determine during what phase each of the various conventional operations should occur, and generates the Static-IL program with the corresponding n s. The examples in Section 4.4, and in particular the two examples in Sections 4.4.3 and 4.4.9, demonstrate what happens in successive phases, how these language constructs work, and the purpose of these n parameters.

4.2.3.1. (*deep-const* c t n)

This construct always returns an ERT. Its purpose is to generate a type-checked quoted constant in the proper phase, i.e. a Static-IL program of the form $\text{'ev. } C$ is any constant value, t is its type, and n is the number of phases to wait before the constant is needed. *Deep-const* can be thought of as deeply quoting the constant. (Constants are not assumed to be self-quoting.)

The operation *deep-const* is evaluated as follows. If $n > 0$, the ERT $\langle (\text{deep-const } c \ t \ n-1), \langle \rangle, \text{ert} \rangle$ is returned; otherwise, (when $n = 0$), the ERT $\langle 'c, \langle \rangle, t \rangle$ is returned. The idea is that each time *deep-const* is evaluated, it basically just decrements n , returning the same kind of ERT until n reaches 0. When n reaches 0, then an ERT containing the quoted constant and its type is returned.²³

4.2.3.2. (*check-funtype* expr_d expr_r n)

Check-funtype always returns an ERT. It is used to generate an ERT containing a funtype expression as its Expression component, when n is 0.

Both expr_d and expr_r will evaluate to ERTs; call them $\langle e_d, t_d, t_d \rangle$ and $\langle e_r, t_r, t_r \rangle$.

Let us first consider the case when the number n is 0, in which case a funtype ERT will be returned. Both t_d and t_r must be the type constant

²³Note that the constant's type is hidden until the phase before the constant is used, even though the type is determined syntactically by the original Static-Phi program. This means that if one type of constant is written where some other type is required, the type mismatch will not be discovered until the phase before the value of the constant would have been used, even though it certainly would be better to report the error as early as possible. This issue is mentioned further in Section 7.2.8.

`type`, indicating that e_r and e_d will evaluate to types during the next phase. (It is a "compiletime" error if either t_d or t_r are not `type`.) Next, the Expression components e_d and e_r are used to build the Expression component of the ERT that `check-funtype` will return. For example, if e_d and e_r are `'number` and `'number`, `check-funtype` will return an ERT with the expression component (`funtype 'number 'number`).

Similarly, the resulting ERT's Required-environment is formed by combining the Required-environments r_d and r_r . This means that the free variables of the resulting expression include the free variables of both of its subexpressions e_d and e_r . However, the Required-environments must be *consistent*: if a variable appears in both, it must have the same type, otherwise it is a "compiletime" error.

Finally, the Type component of the resulting ERT will be `type -- funtype` always returns a type.

If $n > 0$, then this is not the right phase to generate a `funtype` expression; instead, another `check-funtype` expression will be generated, and n will be decremented, as for `deep-const`. In this case, both t_d and t_r must be the type constant `ert`, indicating that e_r and e_d will evaluate to ERTs during the next phase. The resulting ERT will be constructed in a manner similar to the case when a `funtype` expression is generated, except that the Type component of the resulting ERT will be `ert` instead of `type`.

4.2.3.3. (check-lambda $id\ expr_d\ expr_r\ expr_{body}$)

The `check-lambda` construct is analogous to the `check-funtype` construct: it is used to generate a `lambda` Static-IL expression, and it always returns an ERT. However, `check-lambda` differs from `check-funtype` in two important ways: it has a bound variable, id ; and two of its subexpressions, $expr_d$ and $expr_r$, evaluate to types, while the other, $expr_{body}$, evaluates to an ERT.

The `check-lambda` construct is evaluated as follows. First, the type expressions $expr_d$ and $expr_r$ are evaluated in the current environment; call the resulting types t_d and t_r .

Next, an ERT $\langle id, \langle id, t_d \rangle, t_d \rangle$ is formed for the bound variable and its type. The Expression component is simply the formal parameter; the Type component is the function's domain type (the type of the formal parameter); and the Required-environment lists only the formal parameter. This ERT will be used in type checking the body of the function.

Now the body expression $expr_{body}$ is evaluated in an environment augmented by the binding of id to the ERT $\langle id, \langle id, t_d \rangle, t_d \rangle$, and the result is a (type-checked) ERT $\langle e_{body}, r_{body}, t_{body} \rangle$. To verify that the function body really does return the declared type, the Static-IL Machine must have $t_{body} = t_r$; it is a "compiletime" error if they are not equal.²⁴

²⁴Most languages would not actually require these types to be identical, but would instead require only that t_{body} be a type that is *coercible* to t_r . Such gratuitous type conversions do not make a language fundamentally more powerful when the programmer could just as well explicitly call standard type-conversion functions as needed. Coercions are simply provided for convenience.

Because Static-Phi allows the programmer to write functions on ERTs -- like macros -- it is possible that the body expression references a variable that has the same name as the formal parameter, *id*, but a different type. (This is discussed and illustrated in Section 4.2.1.) Therefore, to ensure that any free instances of *id* in the body really are the declared type, *id* is looked up in the required-environment r_{body} to verify that its type is t_d .

Finally, the Static-IL Machine constructs the ERT that is returned by `check-lambda`. The Expression component is $(\text{lambda } id\ e_{\text{body}})$. The Required-environment component is just the required-environment from the body r_{body} , with the formal parameter, *id*, removed.²⁵ The Type component is $\langle \text{fun } t_d, t_r \rangle$.

4.2.3.4. `(check-check-lambda id expr_d expr_r expr_body n)`

This construct is used to generate a `check-lambda` IL expression. Subexpressions $expr_d$, $expr_r$, and $expr_{\text{body}}$ all evaluate to ERTs; an ERT is always returned. `check-check-lambda` is analogous to `check-funtype` in that it waits for the phase when $n = 0$ before generating and returning a `check-lambda` expression. For other phases when $n > 0$, it just decrements n and returns another `check-check-lambda` expression in the resulting ERT.

Recall that the purpose of the `check-lambda` construct is to generate type-checked `lambda` expressions. Similarly, `check-check-lambda` is provided for generating type-checked `check-lambda` expressions. Remember that every expression must be guaranteed type correct during the phase before it

²⁵The formal parameter *id* is a free variable in the function body, but looking from outside at the entire `lambda` expression, it is bound by the `lambda`.

is executed. But notice that two of the arguments to `check-lambda` are *assumed* to evaluate to types. `check-check-lambda` does the type checking necessary to guarantee that those two arguments will indeed evaluate to types.

At this point, the question usually arises as to whether further `check-check-check-` or `check-check-check-check-lambda` constructs might be needed. Fortunately, they are not, and the reason is that for `check-check-lambda`, all evaluated arguments evaluate to ERTs, and the Static-Phi Translator ensures that every expression will initially evaluate to an ERT. That is, the purpose of `check-check-lambda` is to ensure that all of `check-lambda`'s evaluated arguments will indeed be the expected types, and it is required because two of `check-lambda`'s arguments must be type expressions. But all of `check-check-lambda`'s evaluated arguments must be ERTs. And since the Static-Phi Translator only generates expressions that are guaranteed to evaluate to ERTs, no further `check-check-check-lambda` is needed to ensure that the evaluated arguments to `check-check-lambda` will be ERTs.

`check-check-lambda` is evaluated as follows. Subexpressions $expr_d$ and $expr_r$ are evaluated to ERTs. If $n > 0$, their type components must be `ert`; otherwise (when $n = 0$), their type components must be `type`. Next, an ERT is constructed from the formal parameter, id , for use in type checking the body. This is similar to `check-lambda`, except that the type of id is always `ert`. As with `check-lambda`, the body expression $expr_{body}$ is evaluated to an ERT in an environment augmented by this binding. If this ERT's required-environment lists the formal parameter id , its type should be `ert`. Finally, the return ERT is constructed from the Expression and

Required-environment components of the ERTs obtained from evaluating `check-check-lambda`'s subexpressions. If $n > 0$, n is decremented and another `check-check-lambda` is generated for the Expression component; otherwise (when $n = 0$), a `check-lambda` is generated for the Expression component. In either case, the Type component is `ert`. The Required-environment component is generated by combining the subexpressions' required environments, with the formal parameter removed. However, the Static-IL Machine must first ensure that these required-environments are *compatible*: any identifier listed in any of the required-environments must be listed with the same type in each of the subexpressions' required-environments.

4.2.3.5. (`check-apply` $expr_f$ $expr_x$)

This construct is used to generate an `apply` Static-IL expression. Subexpressions $expr_f$ and $expr_x$ both evaluate to ERTs; an ERT is returned.

This construct is different from the others in that the Static-Phi Translator does not determine, in advance, the phase in which a function application will actually occur. Instead, the `check-apply` operation monitors the Type component from its first argument to see when it will become a function rather than an ERT. If it will be a function, the function's domain type is checked against the type of the actual parameter; otherwise, another `check-apply` is generated.

`check-apply` is evaluated as follows. First, subexpressions $expr_f$ and $expr_x$ are evaluated to ERTs $\langle e_f, r_f, t_f \rangle$ and $\langle e_x, r_x, t_x \rangle$. If t_f is a `fun` type, $\langle \text{fun } t_d, t_r \rangle$, then t_d must equal t_x , and an `apply` ERT is returned with Type component t_r . Otherwise, t_f must be `ert` (it is a "compiletime" error if it is

not), and another `check-apply` ERT is returned with Type component `ert`. In either case, the Required-environment component of the resulting ERT is formed by combining r_f and r_x , which must be consistent. Finally, it is a compiletime error if t_f is `ert` but t_x is not `ert`, because this means that the function argument would evaluate to some final basic value (such as a number or `boolean`) in the next phase, whereas the function expression will evaluate to another ERT.

4.2.4. Efficiency of Static-IL

Given that the Static-IL language includes both compiletime and runtime operations, how efficiently can it be processed? Must it be less efficient than a conventional lambda calculus? Might it be more efficient? Without focusing on the details of any specific implementation, we can make some general observations about Static-IL's inherent efficiency. Since the Static-IL Machine is used both for compiletime and runtime, let us examine these roles separately.

On one hand, when the Static-IL Machine is playing the role of runtime, the operations performed are just the simple operations listed in Section 4.2.2. These are in fact identical to the operations of a conventional untyped lambda calculus, and hence can be just as efficiently processed.

On the other hand, when the Static-IL Machine is playing the role of compiletime, it may be more efficient than a conventional compiler because it can use the basic operations of the runtime machine directly instead of simulating them. For example, constant expressions are evaluated directly at compiletime by our single Static-IL Machine, whereas a conventional

compiler must evaluate them by simulating the action of the runtime machine.

Hence, the Static-IL Machine can be just as efficient at performing runtime operations as a conventional runtime machine, and may be more efficient at performing compiletime operations than a conventional compiler.

4.3. Environments for Static-IL Programs

As discussed in Section 2.2.5, environments supply bindings of identifiers to values, for all of a program's free variables. In our simple model, the environment provides the input for a Static-IL program, and a separate environment must be provided for each phase used. This section provides some insight into the purpose of these environments.

4.3.1. Static-Phi Program X

Let us begin by considering a Static-Phi program consisting of only the single identifier, x . This Static-Phi program will be translated to the ERT $\langle x, \langle x, \text{ert} \rangle, \text{ert} \rangle$. The Expression component is simply x ; the Required-environment component is $\langle x, \text{ert} \rangle$, meaning that the only free variable in the expression is x and its type is ert ; and the Type component is ert , because the expression x will evaluate to an ERT.

In order to evaluate the Expression component x we must provide an environment that satisfies²⁶ the Required-environment. In this case, the

²⁶*Satisfies* is defined in Section 2.2.2.

environment must include a value of type `ert` for `x`. As mentioned in Section 3.3.4, every identifier starts out as type `ert`, that is, during the first phase, every identifier must be bound to an ERT. Let us consider some of the possible ERT values that we might provide for `x`.

Suppose we supplied the ERT value $\langle x, \langle x, \text{ert} \rangle, \text{ert} \rangle$ as the value of `x` in the environment. Then, in the first phase, the expression `x` would simply evaluate to this value -- $\langle x, \langle x, \text{ert} \rangle, \text{ert} \rangle$. But this is precisely the ERT that resulted from translating the original Static-Phi program! In effect, `x` has simply evaluated to itself. This is known as the *default ERT* for `x`.

4.3.2. Definition: Default ERT

For any identifier `id`, the ERT $\langle id, \langle id, \text{ert} \rangle, \text{ert} \rangle$ is called the *default ERT* for this identifier.

4.3.3. The Purpose of Default ERTs

Default ERTs are used to pass identifiers through some number of phases before fixing their types. (*Fixing* an identifier's type is discussed below in Section 4.3.4.) They are called "default" ERTs because a command interpreter would normally provide a default ERT binding for each identifier of type `ert` that was not to be fixed to some other type.

4.3.4. Fixing the Type of an Identifier

Suppose that we supply a slightly different ERT value for `x` in the environment: $\langle x, \langle x, \text{number} \rangle, \text{number} \rangle$. This looks similar to the default ERT, but the type of `x` is given as `number` in the Required-environment, and Expression's result type is then `number` also. If we supply this ERT as the

value for `x` in the environment, then, of course, `x` evaluates to this ERT -- `<x, <x,number>, number>`. In this case, even though the Expression component is again `x`, the type of `x` is now given as `number`, that is, `x` must be bound to a `number` in the next phase. Whereas the default ERT simply caused `x` to evaluate to itself, this ERT *fixes* the type of `x` to be type `number`.

4.3.5. Fixing an Identifier as a Function Type

The last example fixed `x` as type `number`. We could just as well fix it to be some function type. For example, if we provided the following ERT value for `x` in the environment,

```
<x, <x,(fun number number)>, (fun number number)>
```

then in the next phase, `x` must be bound to some function from `numbers` to `numbers`.

4.3.6. Fixing an Identifier as a Macro

We have just showed how the type of `x` could be fixed as a function from `numbers` to `numbers`. If we instead fixed the type of `x` as a function from ERTs to ERTs, by supplying the following ERT value for `x` in the environment,

```
<x, <x,(fun ert ert)>, (fun ert ert)>
```

then `x` would act as a macro in the next phase. That is, in the next phase, `x` would be bound to some function that takes code (an ERT) and produces code (an ERT) as its result.

4.3.7. Fixing the Value of an Identifier

In the last three examples, x was bound to an ERT that fixed the type of x for the next phase, requiring x to be bound to some `number` or a function during the next phase. Thus, the *type* of x was fixed for the next phase, but the *value* of x was not fixed for the next phase. Suppose we had instead bound x to the ERT $\langle 5, \langle \rangle, \text{number} \rangle$. In this case, not only is the type fixed for the next phase, but the value is a constant: 5. Since the Expression component of this ERT has no free variables, the identifier x does not even appear in the Required-environment.

4.3.8. Other Possibilities

Of course, these are not the only interesting ERT values that might be bound to x . For example, suppose the ERT $\langle y, \langle y, \text{ert} \rangle, \text{ert} \rangle$ were provided as the value of x . This ERT is identical to the default ERT for x except that it uses the identifier y instead. This, in effect, renames x to y for the next phase.

So far we have discussed some of the ERTs that might be supplied in the environment as values of a Static-IL program's free variables. Of course, free variables of other types, such as `number` or `(fun number number)`, would have to be bound to values of those types in the environment.

4.3.9. What Values to Supply in What Phases

Since a different environment is supplied for each phase, the question arises as to what each of these environments should include. Of course, the Required-environment specifies the *types* of the values that must be provided in the environment, but it does not tell the *purposes* of these values.

In particular, there were several different kinds of ERT values discussed above that might be used for an identifier of type `ert`. How do we know which is appropriate?

The answer depends on the program and the programmer's intent. Every program will be expecting certain kinds of input, via the environment, in certain phases. Information on the kind of input expected in each phase (other than its type) must be provided as external documentation, in the same way that the purposes of any conventional program's inputs must be documented.

There is, however, a pattern to the types of the free variables that one would generally expect to see in various phases. Since every identifier starts out type `ert` (as mentioned in Section 3.3.4), the default ERT would initially be used for that identifier. Then, during some phase the type of this identifier will be fixed to some basic (non-ERT) type, as described above, and finally during the next phase the identifier will have a value of that basic type. Thus, pertinent documentation on this identifier should specify during what phase its type should be fixed.

The examples in Section 4.1.4 help provide an understanding of how various phases are used and what happens in each phase.

4.4. Examples of Translation and Evaluation

This section shows examples of translating all of the Static-Phi programs shown in Section 4.1.4 to Static-IL programs, and executing the resulting Static-IL programs through phases. The most straight-forward and informative examples with which to begin are the two that involve creating

and applying an identity function in Sections 4.4.3 and 4.4.9. The Static-Phi Translator and Static-IL Machine are formally defined in Appendix A.

In the examples below, ERT values $\langle e, r, t \rangle$ are displayed in a LISP-like form:

(ert)

Similarly, environments are displayed as LISP-like lists. Each element of the environment lists an identifier-value binding, which is in turn displayed as a LISP-like list, for example:

(
 $(id_1 value_1)$
 $(id_2 value_2)$
 \dots
 $(id_n value_n)$
)

Finally, Required-environments are also displayed as LISP-like lists. Each element of the Required-environment lists an identifier-type pair, which is in turn displayed as a LISP-like list, for example:

(
 $(id_1 type_1)$
 $(id_2 type_2)$
 \dots
 $(id_n type_n)$
)

4.4.1. F Twice

This example is complicated by the need for a (non-empty) environment. Therefore, the reader is advised to first study the example of Section 4.4.3, Identity Application, which involves no free variables.

(f (f x))

```
----- Result of Translation -----
(
  (check-apply f (check-apply f x))      ; Expression
  ((f ert) (x ert))                      ; Req-env
  ert                                     ; Type
)

----- Environment for Phase 1 -----
(
  (x (x ((x number)) number))
  (f (f ((f (fun number number))) (fun number number)))
)

----- Result of Phase 1 -----
(
  (apply f (apply f x))                  ; Expression
  ((f (fun number number)) (x number))   ; Req-env
  number                                  ; Type
)

----- Environment for Phase 2 -----
(
  (x      5)
  (f      (closure z (incr z) ()))
)

----- Result of Phase 2 -----
7
```

4.4.2. Identity Abstraction

See the example of Identity Application in Section 4.4.3.

$\lambda x : \text{number} \rightarrow \text{number} . x$

```
;----- Result of Translation -----  
(  
  (check-check-lambda                ; Expression  
    x  
    (deep-const number type 0)  
    (deep-const number type 0)  
    x  
    0  
  )  
  ()                                  ; Required-environment  
  ert                                 ; Type  
)
```

```
;----- Environment for Phase 1 -----  
( )
```

```
;----- Result of Phase 1 -----  
(  
  (check-lambda x 'number 'number x) ; Expression  
  ()                                  ; Required-environment  
  ert                                 ; Type  
)
```

```
;----- Environment for Phase 2 -----  
( )
```

```

;----- Result of Phase 2 -----
(
  (lambda x x)                ; Expression
  ()                          ; Required-environment
  (fun number number)        ; Type
)

;----- Environment for Phase 3 -----
()

;----- Result of Phase 3 -----
(closure x x ())

```

4.4.3. Identity Application

This is the best of these examples to study first.

The correspondence between the Static-Phi program and the ERT that results from translation (shown below) is as follows. To dispense with the easy parts first, the Required-environment component of the ERT is empty, because there are no free variables, and the Type component is `ert`, indicating that the result of the first phase will be an ERT, as it always is. In the Expression component, the function application of the original Static-Phi program has been translated to a `check-apply` Static-IL construct. The λ abstraction was translated to a `check-check-lambda`, using the formal parameter name; the type constants `number` and `number` were translated to `deep-const` forms, listing the number of phases to wait as 0; and the identifier, `x`, supplied as the function body, was simply translated to itself, `x`. The generated `check-check-lambda` lists the number of phases to wait as 0. Finally, the constant `5` that was given as the actual argument was translated to another `deep-const` form, listing the number of phases to wait as 1. Note that the `deep-consts` generated for the `number` type constants have one

fewer phases to wait than the `deep-const` generated for the function's actual argument, `5`. This is because the type values will be needed to type check the function application, one phases earlier than the function is applied to the constant `5`.

The progression through phases is as follows. During the first phase, the types of the function's type expressions (`number` and `number`) are checked to ensure that they really are type expressions and not, say, numeric expressions. Upon doing this check, the `check-check-lambda` produces a `check-lambda`. During the second phase, these type expressions will be evaluated to the type values `number` and `number` and these types will be used to generate a type-checked `lambda`. In turn, the `check-apply` verifies that the type of the actual argument matches the function's declared formal parameter type, and generates an `apply` form. Finally, in the third phase, the function is applied to the constant `5` to produce a result of `5`.

The original Static-Phi program and the progression through phases are shown below. Compare this example with the example in Section 4.4.9.

----- Static-Phi Program -----

(λ x : number → number . x 5)

----- Result of Translation -----

```
(
  (check-apply                ; Expression:
    (check-check-lambda
      x                        ; Formal parameter
      (deep-const number type 0) ; Domain type
      (deep-const number type 0) ; Range type
      x                        ; Expression body
      0                        ; Phases to wait
    )
    (deep-const 5 number 1)    ; Actual argument
  )
  ()                            ; Required-environment
  ert                          ; Type
)
```

----- Environment for Phase 1 -----

()

----- Result of Phase 1 -----

```
(
  (check-apply                ; Expression
    (check-lambda x 'number 'number x)
    (deep-const 5 number 0)
  )
  ()                            ; Required-environment
  ert                          ; Type
)
```

----- Environment for Phase 2 -----

()

```

;----- Result of Phase 2 -----
(
  (apply (lambda x x) '5)           ; Expression
  ()                                   ; Required-environment
  number                             ; Type
)

```

```

;----- Environment for Phase 3 -----
()

```

```

;----- Result of Phase 3 -----
5

```

4.4.4. Function Abstraction

$\lambda x : \text{number} \rightarrow \text{number} . (\text{succ} (\text{succ } x))$

```

;----- Result of Translation -----
(
  (check-check-lambda x           ; Expression
    (deep-const number type 0)
    (deep-const number type 0)
    (check-apply succ (check-apply succ x))
    0
  )
  ((succ ert))                   ; Req-env
  ert                             ; Type
)

```

```

;----- Environment for Phase 1 -----
(
  (succ (succ ((succ ert)) ert))
)

```

```

;----- Result of Phase 1 -----
(
  (check-lambda x 'number 'number      ; Expression
    (check-apply succ (check-apply succ x))
  )
  ((succ ert))                          ; Req-env
  ert                                    ; Type
)

;----- Environment for Phase 2 -----
(
  (succ (succ ((succ (fun number number))) (fun number number)))
)

;----- Result of Phase 2 -----
(
  (lambda x (apply succ (apply succ x))) ; Expression
  ((succ (fun number number)))          ; Req-env
  (fun number number)                   ; Type
)

;----- Environment for Phase 3 -----
(
  (succ (closure z (incr z) ()))
)

;----- Result of Phase 3 -----
(closure
  x
  (apply succ (apply succ x))
  ((succ (closure z (incr z) ())))
)

```

4.4.5. Function Application

$(\lambda x : \text{number} \rightarrow \text{number} . (\text{succ} (\text{succ} x)) \ 5)$

```
----- Result of Translation -----  
(  
  (check-apply                               ; Expression  
    (check-check-lambda  
      x  
      (deep-const number type 0)  
      (deep-const number type 0)  
      (check-apply succ (check-apply succ x))  
      0  
    )  
    (deep-const 5 number 1)  
  )  
  ((succ ert))                               ; Req-env  
  ert                                         ; Type  
)
```

```
----- Environment for Phase 1 -----  
(  
  (succ (succ ((succ ert)) ert))  
)
```

```
----- Result of Phase 1 -----  
(  
  (check-apply                               ; Expression  
    (check-lambda  
      x  
      'number  
      'number  
      (check-apply succ.(check-apply succ x))  
    )  
    (deep-const 5 number 0)  
  )  
  ((succ ert))                               ; Req-env  
  ert                                         ; Type  
)
```

```
;----- Environment for Phase 2 -----  
(  
  (succ (succ ((succ (fun number number)) (fun number number)))  
)
```

```
;----- Result of Phase 2 -----  
(  
  (apply (lambda x (apply succ (apply succ x))) '5) ; Expression  
  ((succ (fun number number)) ; Req-env  
   number ; Type  
)
```

```
;----- Environment for Phase 3 -----  
(  
  (succ (closure z (incr z) ()))  
)
```

```
;----- Result of Phase 3 -----  
7
```

4.4.6. Higher Order Function Abstraction

$\lambda f: (\text{funtype number number}) \rightarrow \text{number} . (f(f x))$

```
----- Result of Translation -----  
(  
  (check-check-lambda                               ; Expression  
    f  
    (check-funtype  
      (deep-const number type 0)  
      (deep-const number type 0)  
      0  
    )  
    (deep-const number type 0)  
    (check-apply f (check-apply f x))  
    0  
  )  
  ((x ert))                                         ; Req-env  
  ert                                               ; Type  
)
```

```
----- Environment for Phase 1 -----  
(  
  (x      (x ((x ert)) ert))  
)
```

```
----- Result of Phase 1 -----  
(  
  (check-lambda                                     ; Expression  
    f  
    (funtype 'number 'number)  
    'number  
    (check-apply f (check-apply f x))  
  )  
  ((x ert))                                         ; Req-env  
  ert  
)
```

```
;----- Environment for Phase 2 -----  
(  
  (x      (x ((x number)) number))  
)
```

```
;----- Result of Phase 2 -----  
(  
  (lambda f (apply f (apply f x)))      ; Expression  
  ((x number))                          ; Req-env  
  (fun (fun number number) number)     ; Type  
)
```

```
;----- Environment for Phase 3 -----  
(  
  (x      5)  
)
```

```
;----- Result of Phase 3 -----  
(closure f (apply f (apply f x)) ((x 5)))
```

4.4.7. Higher Order Function Application

$(\lambda f: (\text{funtype number number}) \rightarrow \text{number} . (f (f x)) \quad g)$

```
----- Result of Translation -----
(
  (check-apply
    (check-check-lambda
      f
      (check-funtype
        (deep-const number type 0)
        (deep-const number type 0)
        0
      )
      (deep-const number type 0)
      (check-apply f (check-apply f x))
      0
    )
    g
  )
  ((x ert) (g ert))
  ert
)
; Req-env
; Type
```

```
----- Environment for Phase 1 -----
(
  (f      (f ((f ert)) ert))
  (g      (g ((g ert)) ert))
)

```



```

:----- Result of Phase 1 -----
(
  (check-apply                                ; Expression
    (check-lambda
      f
      (funtype 'number 'number)
      'number
      (check-apply f (check-apply f x))
    )
    g
  )
  ((x ert) (g ert))                          ; Req-env
  ert                                         ; Type
)

```

```

:----- Environment for Phase 2 -----
(
  (x (x ((x number)) number))
  (g (g ((g (fun number number))) (fun number number)))
)

```

```

:----- Result of Phase 2 -----
(
  (apply (lambda f (apply f (apply f x))) g) ; Expression
  ((x number) (g (fun number number)))     ; Req-env
  number                                     ; Type
)

```

```

:----- Environment for Phase 3 -----
(
  (x 5)
  (g (closure z (incr z) ()))
)

```

```

:----- Result of Phase 3 -----
7

```

4.4.8. Identity-Function Type Abstraction

$\lambda t : \text{type} \rightarrow \text{ert} . (\text{emit } \lambda x : t \rightarrow t . x)$

```
----- Result of Translation -----  
(  
  (check-check-lambda                               ; Expression  
    t  
    (deep-const type type 0)  
    (deep-const ert type 0)  
    (check-check-lambda x t t x 1)  
    0  
  )  
  ()                                               ; Req-env  
  ert                                             ; Type  
)
```

```
----- Environment for Phase 1 -----  
( )
```

```
----- Result of Phase 1 -----  
(  
  (check-lambda t 'type 'ert                       ; Expression  
    (check-check-lambda x t t x 0)  
  )  
  ()                                               ; Req-env  
  ert                                             ; Type  
)
```

```
----- Environment for Phase 2 -----  
( )
```

```

;----- Result of Phase 2 -----
(
    (lambda t (check-lambda x t t x))      ; Expression
    ()                                       ; Req-env
    (fun type ert)                          ; Type
)

;----- Environment for Phase 3 -----
()

;----- Result of Phase 3 -----
(closure t (check-lambda x t t x) ())

```

4.4.9. Identity-Function Type Application

This example requires one more phase than the example of Section 4.4.3. If we view what happens in the various phases in terms of the original Static-Phi program, phase 1 checks the types of the type expressions in the outer λ abstraction; phase 2 type checks the outer λ abstraction and its application, and the types of the type expressions in the inner λ abstraction; phase 3 applies the outer function to the actual argument and produces an ERT for a type-checked identity function on **numbers**; and during phase 4 this identity function becomes an actual function value, or closure, that could have been applied to a numeric argument.

;------ Static-Phi Program -----

($\lambda t : \text{type} \rightarrow \text{ert} . (\text{emit } \lambda x : t \rightarrow t . x)$ number)

;------ Result of Translation -----
(
 (check-apply
 (check-check-lambda
 t
 (deep-const type type 0)
 (deep-const ert type 0)
 (check-check-lambda x t t x 1)
 0
)
 (deep-const number type 1)
)
 ()
 ert
);
 : Expression
 : Req-env
 : Type

;------ Environment for Phase 1 -----
(
)

;------ Result of Phase 1 -----
(
 (check-apply
 (check-lambda
 t
 'type
 'ert
 (check-check-lambda x t t x 0)
)
 (deep-const number type 0)
)
 ()
 ert
);
 : Req-env
 : Type

```
;----- Environment for Phase 2 -----  
( )
```

```
;----- Result of Phase 2 -----  
(  
  (apply (lambda t (check-lambda x t t x)) 'number) ; Expression  
  () ; Req-env  
  ert ; Type  
)
```

```
;----- Environment for Phase 3 -----  
( )
```

```
;----- Result of Phase 3 -----  
(  
  (lambda x x) ; Expression  
  () ; Req-env  
  (fun number number) ; Type  
)
```

```
;----- Environment for Phase 4 -----  
( )
```

```
;----- Result of Phase 4 -----  
(closure x x ( ))
```

4.4.10. General Type Abstraction

$\lambda t : \text{type} \rightarrow \text{ert} . (\text{emit } \lambda f : (\text{funtype } t \ t) \rightarrow t . (f (f \ x)))$

```
----- Result of Translation -----
(
  (check-check-lambda                                ; Expression
    t
    (deep-const type type 0)
    (deep-const ert type 0)
    (check-check-lambda
      f
      (check-funtype t t 1)
      t
      (check-apply f (check-apply f x))
      1
    )
    0
  )
  ((x ert))                                          ; Req-env
  ert                                              ; Type
)

----- Environment for Phase 1 -----
(
  (x      (x ((x ert)) ert))
)
```

```

;----- Result of Phase 1 -----
(
  (check-lambda                                     ; Expression
    t
    'type
    'ert
    (check-check-lambda
      f
      (check-funtype t t 0)
      t
      (check-apply f (check-apply f x))
      0
    )
  )
  ((x ert))                                       ; Req-env
  ert                                             ; Type
)

```

```

;----- Environment for Phase 2 -----
(
  (x      (x ((x ert)) ert))
)

```

```

;----- Result of Phase 2 -----
(
  (lambda                                     ; Expression
    t
    (check-lambda
      f
      (funtype t t)
      t
      (check-apply f (check-apply f x))
    )
  )
  ((x ert))                                       ; Req-env
  (fun type ert)                                   ; Type
)

```

```

;----- Environment for Phase 3 -----
(
  (x      (x ((x number)) number))
)

```

```

;----- Result of Phase 3 -----
(closure
  t
  (check-lambda f (funtype t t) t
    (check-apply f (check-apply f x))
  )
  ((x (x ((x number)) number)))
)

```

4.4.11. General Type Application

$(\lambda t : \text{type} \rightarrow \text{ert} . (\text{emit } \lambda f : (\text{funtype } t \ t) \rightarrow t . (f(f \ x))) \ \text{number})$

```

;----- Result of Translation -----
(
  (check-apply                                     ; Expression
    (check-check-lambda
      t
      (deep-const type type 0)
      (deep-const ert type 0)
      (check-check-lambda
        f
        (check-funtype t t 1)
        t
        (check-apply f (check-apply f x))
        1
      )
      0
    )
    (deep-const number type 1)
  )
  ((x ert))                                       ; Req-env
  ert                                             ; Type
)

```



```

;----- Environment for Phase 1 -----
(
  (x      (x ((x ert)) ert))
)

```

```

;----- Result of Phase 1 -----
(
  (check-apply                                     ; Expression
    (check-lambda
      t
      'type
      'ert
      (check-check-lambda
        f
        (check-funtype t t 0)
        t
        (check-apply f (check-apply f x))
        0
      )
    )
    (deep-const number type 0)
  )
  ((x ert))                                       ; Req-env
  ert                                           ; Type
)

```

```

;----- Environment for Phase 2 -----
(
  (x      (x ((x ert)) ert))
)

```

```

;----- Result of Phase 2 -----
(
  (apply                                     ; Expression
    (lambda
      t
      (check-lambda
        f
        (funtype t t)
        t
        (check-apply f (check-apply f x))
      )
    )
    'number
  )
  ((x ert))                                  ; Req-env
  ert                                         ; Type
)

```

```

;----- Environment for Phase 3 -----
(
  (x (x ((x number) number)))
)

```

```

;----- Result of Phase 3 -----
(
  (lambda f (apply f (apply f x)))          ; Expression
  ((x number))                              ; Req-env
  (fun (fun number number) number)         ; Type
)

```

```

;----- Environment for Phase 4 -----
(
  (x 5)
)

```

```

;----- Result of Phase 4 -----
(closure f (apply f (apply f x)) ((x 5)))

```

4.4.12. Macro Abstraction

$\lambda m : \text{ert} \rightarrow \text{ert} . (m (m x))$

```
----- Result of Translation -----  
(  
  (check-check-lambda                               ; Expression  
    m  
    (deep-const ert type 0)  
    (deep-const ert type 0)  
    (check-apply m (check-apply m x))  
    0  
  )  
  ((x ert))                                         ; Req-env  
  ert                                               ; Type  
)
```

```
----- Environment for Phase 1 -----  
(  
  (x      (x ((x ert)) ert))  
)
```

```
----- Result of Phase 1 -----  
(  
  (check-lambda m 'ert 'ert                          ; Expression  
    (check-apply m (check-apply m x))  
  )  
  ((x ert))                                           ; Req-env  
  ert                                                 ; Type  
)
```

```
----- Environment for Phase 2 -----  
(  
  (x      (x ((x ert)) ert))  
)
```

```

;----- Result of Phase 2 -----
(
  (lambda m (check-apply m (check-apply m x))) ; Expression
  ((x ert)) ; Req-env
  (fun ert ert) ; Type
)

```

```

;----- Environment for Phase 3 -----
(
  (x (x ((x ert)) ert))
)

```

```

;----- Result of Phase 3 -----
(closure
  m
  (check-apply m (check-apply m x))
  ((x (x ((x ert)) ert)))
)

```

4.4.13. Macro Application

```
(λ m : ert → ert . (m (m x))
  (emit λ y : number → number . (succ (succ y)))
)
```

```
----- Result of Translation -----
(
  (check-apply                               ; Expression
    (check-check-lambda
      m
      (deep-const ert type 0)
      (deep-const ert type 0)
      (check-apply m (check-apply m x))
      0
    )
    (check-check-lambda
      y
      (deep-const number type 1)
      (deep-const number type 1)
      (check-apply succ (check-apply succ y))
      1
    )
  )
  ((x ert) (succ ert))                       ; Req-env
  ert                                         ; Type
)

```

```
----- Environment for Phase 1 -----
(
  (succ (succ ((succ ert)) ert))
  (x    (x ((x ert)) ert))
)

```

```

;----- Result of Phase 1 -----
(
  (check-apply                               ; Expression
    (check-lambda
      m
      'ert
      'ert
      (check-apply m (check-apply m x))
    )
    (check-check-lambda
      y
      (deep-const number type 0)
      (deep-const number type 0)
      (check-apply succ (check-apply succ y))
      0
    )
  )
  ((x ert) (succ ert))                       ; Req-env
  ert                                         ; Type
)

```

```

;----- Environment for Phase 2 -----
(
  (succ (succ ((succ ert)) ert))
  (x (x ((x ert)) ert))
)

```

```

;----- Result of Phase 2 -----
(
  (apply                                     ; Expression
    (lambda m (check-apply m (check-apply m x)))
    (check-lambda
      y
      'number
      'number
      (check-apply succ (check-apply succ y))
    )
  )
  ((x ert) (succ ert))                       ; Req-env
  ert                                         ; Type
)

```

```

;----- Environment for Phase 3 -----
(
  (x (x ((x number)) number))
  (succ (succ ((succ (fun number number)) (fun number number)))
)

```

```

;----- Result of Phase 3 -----
(
  (apply
    (lambda y (apply succ (apply succ y)))
    (apply (lambda y (apply succ (apply succ y))) x)
  )
  ((succ (fun number number)) (x number)) ; Req-env
  number ; Type
)

```

```

;----- Environment for Phase 4 -----
(
  (x 5)
  (succ (closure z (incr z) ()))
)

```

```

;----- Result of Phase 4 -----
9

```

Chapter 5

Using Phases for Partial Evaluation

This chapter describes how phases might be used to perform partial evaluation. In this approach, the phase for a particular subcomputation is not denoted in the source program or determined when the program is translated, but is determined dynamically by the environment supplied for each phase. This approach has not been fully explored, and is open for future research, but we demonstrate how it might proceed by describing a source language, *Dynamic-Phi*, and the beginnings of an implementation language, *Dynamic-IL*. There is no formal semantics given for these languages, since they are not fully developed.

5.1. The Dynamic-Phi Language

The Dynamic-Phi language is identical to the Static-Phi language described in Section 4.1, except it does not provide the `emit` and `eval` constructs or the type constant `ert`; hence Dynamic-Phi is not discussed further here. Instead of allowing the programmer to explicitly manipulate ERTs under program control, the system uses ERT values transparently, to represent the results of a partial evaluation.

5.2. The Dynamic-IL Language

As with the Static-IL language, Dynamic-IL looks like an untyped lambda calculus because type declarations are not explicit, but in fact it is strongly typed. In fact, almost all of the constructs of Static-IL and Dynamic-IL look similar, though the semantics of constructs that manipulate ERT values are necessarily different, as discussed in Section 5.2.2.

5.2.1. Conventional Lambda Calculus Operations

Dynamic-IL has the following simple operations that look like a conventional lambda calculus written in the LISP [McCarthy 66] style. These function exactly the same as in Static-IL. They are:

ev Any quoted expressible value.

id An identifier.

(*lambda id expr*) Function abstraction.

(*apply expr_f expr_x*)
 Function application.

(*funtype expr_r expr_d*)
 The type of a function. As with Static-IL, the returned function type is represented as a pair, tagged with the word fun: <fun *type_d*, *type_r*>.

These operations are not discussed further here.

5.2.2. Other Operations

All but one of the other operations look similar to operations in Static-IL, except that the operations below lack the n parameter, and hence their semantics are somewhat different. In Static-IL, the n parameter specified how many phases to wait before generating one of the conventional operations, and this was determined statically, during translation. But in Dynamic-IL, the determination of when to generate one of the conventional operations is done dynamically by each of the operations listed below. Compared with Static-IL, Dynamic-IL is missing one operation, `deep-const`, and contains one new operation, `hold`. `Deep-const` is unnecessary because there is no *a priori* determination of when a constant will be needed; `hold` is now used to pass a value computed in one phase, to the next phase.

5.2.2.1. (`hold t expr`)

`hold` always returns an ERT. The argument t may be any value of type `type` -- it is not an expression -- and `expr` is an expression that evaluates to a value of that type. `hold` simply evaluates expression `expr` to some value ev , and returns the ERT $\langle ev, \langle \rangle, t \rangle$. Thus, even though `expr` is evaluated during this phase, its value is not used until the next phase.

`hold` is typically used to synchronize two values that are needed by an operation. Each of the `check-` operations must detect this and generate `holds` as needed. For example, the `funtime` operation has two subexpressions that must evaluate to types. During the phases before the subexpressions evaluate to types, they will evaluate to ERTs. What if one of the subexpressions is ready to evaluate to a type during some phase, but the

other is still going to evaluate to an ERT, and will not evaluate to a type until the following phase? In that case, the subexpression that is ready to evaluate to a type can be evaluated, and the `hold` operation can be used to pass the resulting type value on to the next phase, when the other subexpression will also evaluate to a type. Thus `check-funtype` can force both subexpressions to return ERTs during one phase, and during the next phase, the `funtype` operation will have both type values as needed. Section 5.2.2.2 explains specifically how this works for the `check-funtype` operation. Other `check-`operations work analogously.

5.2.2.2. (`check-funtype` $expr_d$ $expr_r$)

`check-funtype` always returns an ERT; it is used to generate a `funtype` ERT. However, unlike in Static-IL, `check-funtype` will not necessarily generate a `funtype` expression during this phase. If its arguments will not be ready to be fully evaluated to types during the next phase (that is, if its arguments are still going to evaluate to ERTs), another `check-funtype` expression is generated. This is similar to the way `check-apply` in Static-IL generates an `apply` if the function arguments will be ready in the next phase, and a `check-apply` if not.

`check-funtype` is evaluated as follows. Both subexpressions $expr_r$ and $expr_d$ are evaluated to ERTs, call them $\langle e_d, r_d, t_d \rangle$ and $\langle e_r, r_r, t_r \rangle$. If both t_d and t_r are type, a `funtype` expression is generated, as in Static-IL. If both t_d and t_r are `ert`, the returned ERT will contain a `check-funtype` expression: the Expression component will be $(\text{check-funtype } e_d \ e_r)$; the Required-environment component will be the combination of r_d and r_r , which must be consistent (as defined in Section 4.2.3.2); and the Type component will be `ert`.

Note that, for a `funtype` expression to be generated, both t_d and t_r must be `type`, indicating that e_d and e_r will evaluate to types. Since every expression starts out (after translation) evaluating to an ERT, in effect t_d and t_r will start out as `ert` and will become `type` during some later phase. But what if one of the `check-funtype`'s subexpressions is ready to evaluate to a type before the other is ready? That is, what if either t_d or t_r is `type`, but the other is `ert`?

In this case, we can simply allow the type value to be computed during the next phase, but use a `hold` expression to pass the result on to the next phase, to be ready during the phase when the other subexpression is also ready to evaluate to a type. Thus, a `check-funtype` is generated as before, but a `hold` is inserted to pass the type value to a subsequent phase as a constant. For example, if t_d is `type` and t_r is `ert`, the resulting Expression component will be `(check-funtype (hold type e_d) e_r)`.

It is a "compiletime" error if either t_d or t_r is not `type` or `ert`.

5.2.2.3. `(check-lambda id $expr_d$ $expr_r$ $expr_{body}$)`

Analogous to `check-funtype`, `check-lambda` is used to generate a `lambda` but should generate another `check-lambda` if the body expression will not be ready during the next phase (that is, if the body expression evaluates to an ERT whose Type component is `ert`). If another `check-lambda` is generated, the type expressions will simply be the quoted type values that were computed during this phase.

The implementation of `check-lambda` is not as straightforward as it may at first seem; its discussion is postponed to Section 5.3.

5.2.2.4. (check-check-lambda *id expr_d expr_r expr_{body}*)

This construct is handled straightforwardly in a manner analogous to `check-funtype` above. After evaluating $expr_d$ and $expr_r$, an ERT binding (with Type component `ert`) is created for the formal parameter, and the body expression $expr_{body}$ is executed in an environment augmented by this binding. If both $expr_d$ and $expr_r$ have evaluated to ERTs whose Type component is `type`, a `check-lambda` will be generated. Otherwise another `check-check-lambda` should be generated, with `hold` used as necessary.

5.2.2.5. (check-apply *expr_f expr_x*)

This construct is evaluated as in Static-IL, except that `hold` may be inserted as needed if either the function or the argument is ready before the other (that is, if one will still be an ERT when the other will be a function or non-ERT value during the next phase).

5.3. Problems in Implementing Check-Lambda

Before discussing these issues, it should first be noted that there are several ways of partially evaluating function calls. Beckman et al. [Beckman 76] provide a good outline of the various methods. We will restrict our attention to the simplest choice.

Suppose we have the following lambda abstraction in Dynamic-Phi, which has free variable `f`:

$$\lambda x : \text{number} \rightarrow \text{number} . (f x)$$

And consider the corresponding Dynamic-IL program:

```
(check-lambda x 'number 'number (check-apply f x))
```

where *f* is type *ert*.

Now, our goal here is to come up with a method of implementing the *check-lambda* operation. As a general outline, it should proceed according to the following steps:

1. Evaluate the two *type* subexpressions. In this example, they are *'number* and *'number*, and they simply evaluate to the *type* values *number* and *number*.
2. Decide on a suitable ERT binding for the formal parameter, and add this binding to the environment. In our example, we must bind *x* to the proper ERT, and add this binding to the environment. (An ERT binding for *f* will already be in the environment.)
3. Evaluate the body expression to an ERT in this new environment. In our example, we must evaluate the *check-apply* to an ERT.
4. Using the ERT that resulted from evaluating the body expression, construct and return either a *lambda* ERT, if the body is ready to be evaluated to some basic non-ERT value in the next phase; or a *check-lambda*, if the body must evaluate to another ERT in the next phase.

Thus, in our example, the result of step 3 will either be an *apply* ERT (if the body is ready to evaluate to a *number*), such as the following (call this *ertA*):

```
(
    ; ertA
    (apply f x)           ; Expression
    ((f (fun number number))) ; Required-environment
    number                ; Type
)
```

or a `check-apply` ERT (if the body must still evaluate to another ERT), such as the following (call this *ertB*):

```
(
    ; ertB
    (check-apply f x)      ; Expression
    ((f ert))             ; Required-environment
    ert                   ; Type
)
```

The `Type` component of *ertA* indicates that it is ready to evaluate to a `number`, whereas the `Type` component of *ertB* indicates that it will evaluate to another ERT.

Finally, the result of evaluating the `check-lambda` (i.e. the result of step 4) should either be a `lambda` ERT, such as the following (call this *ert1*):

```
(
    ; ert1
    (lambda x (apply f x)) ; Expression
    ((f (fun number number))) ; Required-environment
    (fun number number)   ; Type
)
```

or it should be a `check-lambda` ERT, such as the following (call this *ert2*):

```
(
    ; ert2
    (check-lambda x (check-apply f x)) ; Expression
    ((f ert))                          ; Required-environment
    ert                                 ; Type
)
```

That is, the result of step 4 should be *ert1* if the result of step 3 is *ertA*, whereas it should be *ert2* if the result of step 3 is *ertB*. That much is straightforward. The difficulty is this: What ERT binding should we provide for `x` in step 2?

If the result of step 3 will be *ertA*, then we should supply a binding of `x` to

the ERT $(x (x \text{ number}) \text{ number})$ in step 2, since x the function can be applied to a number in the next phase, and hence x will be bound to a number in the next phase. This binding, in effect, declares x to be type `number` in the next phase.

On the other hand, if the result of step 3 will be *ertB*, then we should supply a binding of x to the ERT $\langle x, \langle x, \text{ert} \rangle, \text{ert} \rangle$ in step 2, since x another `check-apply` will be evaluated in the next phase, and hence x must be bound to another `ert` in the next phase. This binding, in effect, declares x to be type `ert` in the next phase.

Here is the dilemma. Since the result of evaluating the body in step 3 will in general depend on factors other than just the binding of x (in this case it also depends on the binding of f from outside), we cannot generally know which binding for x to use in step 2 until we know the result of step 3.

In our example, two potential values for f that would cause different results would be the ERT:

```
(
    f                ; Expression
    ((f (fun number number))) ; Required-environment
    (fun number number) ; Type
)
```

and the ERT:

```
(
    f                ; Expression
    ((f ert))        ; Required-environment
    ert              ; Type
)
```


5.3.1. Evaluating the Body Twice

One way to deal with this dilemma might be to first assume that the `check-lambda`'s body expression will evaluate to an ERT that will be ready to be "fully" evaluated during the next phase; that is, first assume that step 3 will evaluate to an ERT such as *ertA*, in which the Type component is not `ert`. Thus, we would initially bind `x` to the ERT $\langle x, \langle x, \text{number} \rangle, \text{number} \rangle$. If the Type component of the result of step 3 turns out to be a basic (no-ERT) type, such as *ertA*, then all is well, and the result of `check-lambda` should be a `lambda` ERT, such as *ert1*. However, if the Type component turns out to be `ert`, such as in *ertB*, then we bind `x` to the ERT $\langle x, \langle x, \text{ert} \rangle, \text{ert} \rangle$ and re-evaluate the body as in step 3 again.

5.3.2. Evaluating with Both Choices at Once

A more efficient solution might be to have the execution of $\text{expr}_{\text{body}}$ generate the two ERTs, under both assumptions. But what happens when functions are nested? Must 4, 8, etc., ERTs be generated? When can the choices be eliminated?

5.3.3. Using an Extra Environment Variable

Since the difficulty seems to be in deciding which binding to use for the formal parameter, another possibility might be for the Dynamic-IL Machine to use an extra environment variable while a `check-lambda` body is being executed, indicating the names of any identifiers that are bound as formal parameters. Those identifiers could then be treated specially by the Dynamic-IL Machine. This might allow the body to be evaluated in one pass. But notice, then, that the machine would essentially be playing the

role of two distinct machines, depending on whether this extra environment variable were set.

5.3.4. The Root of the Problem

The root of the problem is that we are asking the Dynamic-IL Machine to do two kinds of things during the same phase: to partially and fully *evaluate* certain subexpressions, and to *compile* certain subexpressions for future partial or full evaluation. Section 5.4 proposes another solution.

5.4. A Strongly Typed Phase Compiler

A better approach to the problems of implementing `check-lambda` in Dynamic-IL might be to evaluate programs in two distinct passes: call the first pass *phase compilation* and the second pass *phase evaluation*. As before, phase evaluation would perform both partial and full evaluation, filling the roles of traditional compiletime and runtime.

During **phase compilation** every expression would be treated symbolically: none of the subexpressions would evaluate to a final (constant) value and no type checking would be done. The purpose of phase compilation would be to *decide* which subexpressions can be fully evaluated and which should be partially evaluated. The resulting program will have these decisions syntactically built into it (as with `lambda`, `check-lambda`, etc.), ready for phase evaluation. To make these decisions, the phase compiler must know which of the program's free variables are to be given final (constant) values during phase evaluation.

Phase evaluation could then fully evaluate some subexpressions and

partially evaluate others. The result of the phase evaluation would either be some final constant or another program (ERT), but its type would be known beforehand. The key to this approach is that the type of every expression is known before phase evaluation. Thus, those expressions being fully evaluated can be evaluated just as efficiently as on a conventional (i.e. fully evaluating) abstract machine, even though the phase evaluation machine is also performing partial evaluation for a strongly typed language.

This approach, for strongly typed languages, is analogous to the "compiled generation" approach used by Beckman, et al. [Beckman 76] in partially evaluating LISP programs.

Chapter 6

Remarks About Other Language Notions

This section discusses some miscellaneous language notions, and shows how phases are relevant to them or vice-versa.

6.1. Abstract Data Types

This section discusses one possible approach to providing Abstract Data Types (ADTs) under a model of phases. The purpose of this section is to demonstrate some of the usefulness of manipulating ERTs under a model of phases. The ERT data type makes it easy to talk clearly and sensibly about compiletime notions such as enforcing the information hiding needed to implement ADTs.

Our *notion* of ADTs is intended to be ordinary -- corresponding basically to Ada packages, for example -- but our view of *implementing* ADTs is somewhat unusual, and is motivated by the fact that we treat types and code (ERTs) as first-class values. That is, we intend to provide the same basic functionality of traditional ADTs, but we take an unusual view of what is required and how to provide it. In effect, this discussion treats ADTs from a compiler's point of view, since phases fill the role of traditional compile-time.

We consider a newly defined ADT to be essentially a unique type and a

set of operations that are privileged to operate on values of that type. As in Ada, we assume that an ADT has no separate functional or behavioral specification: its intended behavior is defined only by its implementation. The programmer defines a new ADT in terms of other types and operations, thus supplying an implementation for it. The implementation should be hidden from the user; this information hiding should be enforced by the language.

Since, in a strongly typed language, this information hiding must be enforced by the compiler, and there need not be anything special about the runtime code, ADTs are essentially compiletime notions. Under a model of phases, a phase fills the role of compiletime, manipulating both types and code (ERTs) as first-class values. Therefore, to understand the following discussion, it is best to think of ADTs in terms of what a compiler must do to enforce the required information hiding. To focus only on the essential elements, we do not address scoping rules or other extraneous issues such as providing separate declarations of ADT headers and bodies, as is allowed in Ada.

6.1.1. Four Essential Functions

Let us personify the portion of the program that implements the ADT as the *implementor*, and the portion of the program that uses the ADT as the *user*. To employ the canonical example, we might define an ADT called **stack**, offering only **push** and **pop** functions for accessing values of type **stack**, and use an **array** to implement the stack.²⁷ The compiler, then, must

²⁷Usually, a stack would be implemented by a pair consisting of an array and an integer, with the integer representing a pointer to the current stack top. This detail is irrelevant here, and we are ignoring it for the sake of simplicity.

ensure that the implementation of type **stack** as an **array** is hidden from the user, but is available to the **stack** implementor.

The **stack** implementor, then, must be privileged to perform two essential acts: to create a value of the ADT from a value of the implementing type, for example, creating a **stack** value from an **array** value; and to view a value of the ADT as a value of the implementing type, for example viewing a **stack** value as an **array**. Less obviously, though, in a language in which types are first-class values, the implementor of the **stack** ADT must also be privileged to perform two additional essential acts: to create the type value **stack** from the implementing type value **array**; and to view the type value **stack** as the type value **array**.

All four of these privileged acts are compiletime sleight of hand²⁸ -- they are functions involving types that are computed at compiletime. Recall that during "compiletime" (a relative term), type and ert values are manipulated, and that in our model a phase fills the role of traditional compiletime. Thus, to create a **stack** ADT, we need the following four functions.

abs-stack: ert → ert

For creating values of type **stack** from values of type

²⁸Pascal's *ord* function is probably the best known example of a "compiletime sleight of hand". For any scalar type, the *ord* function returns an integer representing the argument's ordinal value. *Ord* is almost universally implemented in the compiler simply by viewing the binary representation of its argument as a value of a different type. For example in a Pascal system in which the ASCII character set is used, the value of *ord*('X') would be 88. That is, the binary value 1011000 is simply interpreted as representing an integer (88) instead of representing the ASCII character 'X'. Because the binary representations of these values are the same, the compiler does not emit any code to implement the *ord* function. Although the compiler would view the expression 'X' as producing a value of type **char** and the expression *ord*('X') as producing a value of type **integer**, the code generated for these two expressions would be absolutely identical.

array. Note that this function takes an ERT value and returns an ERT value -- it does not take an **array** value and return a **stack** value. Rather, it takes an *expression* (an ERT) that will evaluate to an **array** value, and returns an *expression* (an ERT) that will evaluate to a **stack** value. Only the Type component of the argument ERT and the result ERT will differ.

imp-stack: ert → ert

For viewing values of type **stack** as values of type **array**. Again, this function takes an expression (an ERT) that will evaluate to a **stack** and returns an expression that will evaluate to an **array**.

type-abs-stack: type → type

For creating a **stack** type value from an **array** type value. Note that this is a function that takes a type and returns a type. The type value that is supplied as the actual parameter must be an **array** type value; a **stack** type value will be returned.

type-imp-stack: type → type

For viewing **stack** type values as **array** type values. Note that this is a function that takes a type and returns a type. The type value that is supplied as the actual parameter must be a **stack** type value; an **array** type value will be returned.

6.1.2. Type Values: $\langle tag, value \rangle$ Pairs

Let us now assume that type values are represented as $\langle tag, value \rangle$ pairs.

The *tag* component is a symbol identifying the type; for example, it might be the symbol 'array, representing any array type. It is the same for every array type.

The *value* component holds other information about that particular type; for example, it might include information about the array's element type and size (if the size is a part of the type, as it is in Pascal, for example). The *value* component will generally be different for different array types. For programmer-defined ADTs, it will always be a type.

6.1.3. Type-of and Tag-of

The representations of types and erts should of course be hidden, but to determine the Type component of an ERT, or the Tag component of a type, functions **type-of** and **tag-of** can be provided:

type-of: ert → type

tag-of: type → symbol

If the representations of ERTs and types were visible, these functions would be defined simply:

(type-of <e,r,t>) = t

(tag-of <t,v>) = t

6.1.4. Implementations of the Four Essential Functions

The four essential functions for a **stack** ADT can be roughly defined as follows. Bear in mind that types **type** and **ert** should also be ADTs, and the programmer would not have indiscriminate access to their representations. However, for expository purposes, the functions defined below show the representations of **types** as pairs and **erts** as triplets. (**Error**, below, represents a "compiletime" error condition indicating that the programmer tried to use one of these **stack** functions to convert between something other than a **stack** and an **array**.)


```

(abs-stack <e,r,<t,v>> ) =
  if t = 'array
  then <e,r,<'stack,<t,v>>> -- Save the implementing type.
  else error

(imp-stack <e,r,<t,v>> ) =
  if t = 'stack
  then <e,r,v>             -- Restore the implementing type.
  else error

(type-abs-stack <t,v> ) =
  if t = 'array
  then <'stack,<t,v>>      -- Save the implementing type.
  else error

(type-imp-stack <e,r,<t,v>> ) =
  if t = 'stack
  then v                 -- Restore the implementing type.
  else error

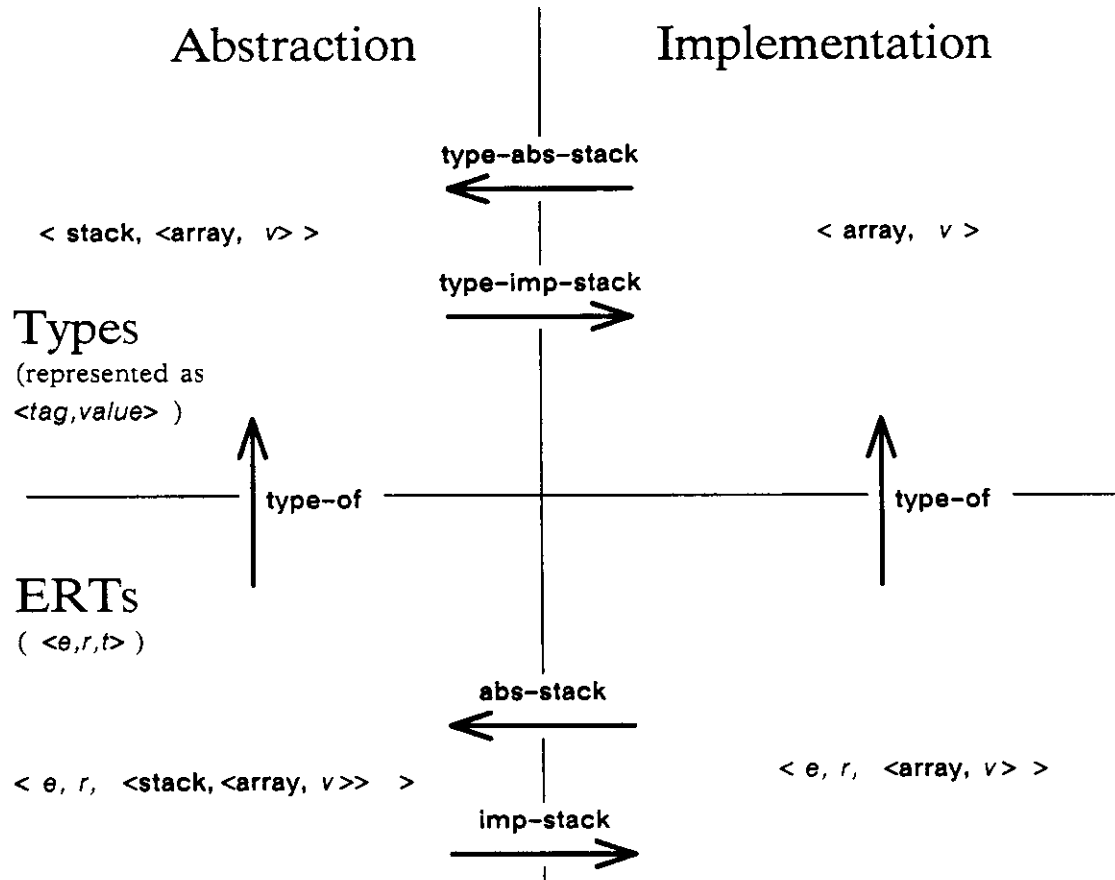
```

The relationships between these four functions are illustrated in Figure 6-1.

6.1.5. Defining a New Abstract Data Type

To allow a new abstract data type to be defined, the language needs to supply a function that will create and return the four functions described above, with a new uniquely generated tag embedded in them. The new tag should also be returned, to allow the programmer to test for this new type without incurring an error.

**Figure 6-1:
Implementing Abstract Data Types**



6.2. Dependent Types

Dependent types, for example, of Pebble [Burstall 84], are compound types in which the type of one element depends on the value of another element. For example, in Pebble a dependent type is used to express the type of a polymorphic pair-swapping function:

$$(t1:\text{type} \times t2:\text{type}) \rightarrow \lambda (t1 \times t2 \rightarrow t2 \times t1)$$

The symbol " $\rightarrow \lambda$ " is similar to " \rightarrow " except that bound variables appear on the left and may be used on the right to refer to their values. This polymorphic swapping function is actually a function that returns a function: it is first given the types of the elements to swap, and the result is then a swapping function, specific to those types, that may be applied to an actual pair of elements. It swaps and returns the first and second elements of the pair. Thus, for example, if we wish to swap `[int, bool]` pairs, returning `[bool, int]` pairs, instantiating *swap* for these types would yield a function value of type `int × bool → bool × int`:

$$\text{swap}[\text{int}, \text{bool}]: \text{int} \times \text{bool} \rightarrow \text{bool} \times \text{int}$$

Dependent types seem to have arisen mainly from the desire to assign sensible types to all expressions, yet also be able to parameterize something by a type, such as a polymorphic function; manipulate type-tagged values at runtime; and define recursive types.

Phi does not offer dependent types, but some of the same functionality could be obtained in other ways, as described below.

In Static-Phi, arbitrary type expressions can be evaluated at compiletime, and polymorphic functions or data structures can be instantiated to particular types. In Pebble [Burstall 84], one is unable to talk about a function's actual parameter without dealing with the parameter's runtime value. But in Static-Phi, a function's actual parameter is an ERT value during the phase before the function is applied, so the Type component can meaningfully be extracted and used at that time. This also means that a polymorphic function need not have an extra explicit type parameter.

Dependent types also allow type-tagged values to be manipulated at runtime, and this may be a desirable capability to provide. This can be accomplished by providing a type `any` -- a variable of type `any` could hold a value of any other type, tagged with the value's type. A case conformity clause can be used to query the variable's current type and access its value while retaining strong typing. (This is essentially the way Algol-68 [Lindsey 71] provides union types.) Note that the current work of Gifford, Schooler, et al. [Schooler 84] takes a very attractive approach to this: where possible, they do type checking before runtime; if a type cannot be determined before runtime, dynamic type checking is used.

Recursive types are discussed in Section 7.2.7.

6.3. Type Checking Recursive Functions

The question usually arises: "Is there any special difficulty in type checking recursive functions?" Not when the function's parameter and return types are declared. In fact, the type checking is very similar to the non-recursive case, even when types are allowed as first-class values.

The idea of a recursive function is that the function's name can be used inside the function body. The only impact this has on type checking is that the function's type must be known inside the body. This is easy to arrange, because the function's type is known from parameter and return type declarations.

Compare the type checking required for a non-recursive `let` construct versus a recursive `letrec` construct. The two constructs would be:

`let $id : expr_{type} = expr_{value}$ in $expr_{body}$`

`letrec $id : expr_{type} = expr_{value}$ in $expr_{body}$`

In each case, $expr_{type}$ gives the type of the bound variable id . Call this type t . The only difference in type checking the two constructs is that in type checking $expr_{value}$ for `letrec`, id is known to be type t , rather than whatever type it may have been declared to be in the surrounding scope. This is true even if the function happens to construct a type value.²⁹

²⁹As mentioned in Section 3.3.2, the type that is constructed can be used as an invariant of the next phase (i.e. it can be used in a declaration pertaining to the next phase), but it cannot be used as an invariant of the phase during which it is computed.

Chapter 7

Conclusions and Future Work

7.1. Conclusions

This work has addressed the basic question of whether types and code can be manipulated as first-class values while retaining strong typing. We demonstrated how this can be done by introducing the notion of multiple strongly typed evaluation *phases*. In the simplest case, two phases correspond to the traditional notions of compiletime and runtime, though a single machine is used for both. In general, multiple phases may be used, and each phase acts as compiletime relative to the next phase, or runtime relative to the previous phase. Types that are freely manipulated as first-class values during one phase become invariants of the next phase, thus guaranteeing that the next phase is strongly typed.

One benefit of allowing types and code to be manipulated as first-class values under the model of phases is that the same abstract machine can be used to both compile and run the program. This means that all of the features that are available in the language at runtime are also available at compiletime. The features only need to be implemented once in the single machine, and they are thus guaranteed to have the same semantics at compiletime and runtime. Thus, for example, constant expressions can be evaluated at compiletime using the same efficient evaluation mechanism as is used at runtime, whereas, in general, a conventional compiler must

simulate the action of the runtime machine in evaluating constant expressions. The single machine is therefore inherently "efficient" in two respects: (1) for runtime operations, it can have the same efficiency as a conventional machine in evaluating untyped lambda calculus expressions, even though it has the additional capability of performing compiletime operations; and (2) for compiletime operations it can be much more efficient than a conventional compiler, because compiletime tasks that can already be performed at runtime, such as evaluating constant expressions, are executed directly rather than being simulated.

The special abstract data type ERT is essential to constructing and manipulating code fragments as first-class values, while capturing all information necessary to ensure that any code generated in this manner will be strongly typed. The ERT data type makes it possible to use the same abstract machine to do the compiletime operations of type checking and code generation, as well as conventional runtime operations. The ERT data type also makes it easy to talk sensibly about compiletime notions such as asking for the type of an expression, or dealing with the type conversions involved in implementing abstract data types.

The notion of phases, with its ERT data type and uniform treatment of compiletime and runtime, gives insight into the semantic processing that occurs during compiletime and runtime. It also gives insight into how to efficiently implement compiletime notions such as type checking, using runtime machinery, and how to efficiently provide runtime notions at compiletime.

We believe that the notions of strong typing, types as first-class values, and

partial or phase evaluation complement each other handsomely in providing a language basis for writing more reusable, correct, and efficient software: "reusable" because types can be manipulated as first-class values, and because of the ability to construct new strongly-typed programs with phases or specialize programs with partial evaluation; "correct" because of strong typing; and "efficient" because of the ability to perform much of the computation before runtime.

The following sections outline some suggested future work.

7.2. Subjects for Further Study

7.2.1. Developing a Practical Language Based on Static-Phi and Static-IL

The particular model of phases embodied in the Static-Phi and Static-IL languages of Chapter 4 are based on typed and untyped versions of the lambda calculus, and were presented as purely pedagogical languages. It would be reasonably straightforward to expand these into useful real-life functional languages with a full complement of data types and operators.

7.2.2. Using Phases for Partial Evaluation

This was discussed in Chapter 5.

7.2.3. Constructing and Maintaining Environments

More work is needed on how to effectively generate and manipulate the environment required for each phase. This comes in the larger context of programming methodology.

7.2.4. Determining the Source of a Bug

Suppose a bug is discovered. Where did it originate? During what phase? To some extent, the difficulty of determining the origin of a bug becomes inherently more difficult with more reusable software, in the following sense. When a program is constructed from several pieces of different origins, it may be more difficult to know which piece of the program is at fault when a bug is discovered. On the other hand, if a standard set of reusable software components are provided, they can be very thoroughly debugged. Overall, we do not know whether multiple phases will make debugging significantly more or less difficult.

7.2.5. Universal Polymorphism

A function is *polymorphic* if different parameter types may be used in different invocations. Burstall and Lampson [Burstall 84] distinguish between two kinds of polymorphism (attributing the distinction to C. Strachey [Strachey 67]):

Ad hoc [or Generic] polymorphism

The code executed depends on the type of the argument, e.g., 'print 3' involves different code from 'print "nonsense"'.

Universal [or Parametric] polymorphism

The same code is executed regardless of the type of the argument, since the different types of data have uniform representation, e.g. *reverse(1,2,3,4)* and *reverse(true,false,false)*.

Ad hoc polymorphism is the natural form of polymorphism under phases. Universal polymorphism seems to require something additional. The basic

difficulty is that, in type checking the call of a universally polymorphic function, such as *reverse* (above), different result types should be returned for calls using different actual parameter types, even though the same function will be called at runtime. Furthermore, a mechanism for type checking the function body once, independent of call types, should be provided.

ML [Gordon 79] uses unification in type checking polymorphic functions. Unification involves having the language processor perform substantial computations involving types. This approach might be used here, although it would seem to be somewhat contrary to the underlying philosophy of having types of expressions simply computed rather than inferred by a more complex language processor. It would be most attractive to use an approach that takes advantage of a language's existing ability to explicitly manipulate types as first-class values, as the Static-Phi language does, rather than adding type inference machinery to the language processor. We do not know how best to do this.

7.2.6. Inferring Types

As explained in Section 1.3.1, this work was motivated by a bias toward expressing rather than inferring. However, much notable work on data types, such as ML [Gordon 79] has involved type inference. It would be good to explore the relationship between type inference systems and our model of multiple phases, which is based on types being computed directly. Maybe a hybrid would be feasible.

7.2.7. Recursive Types

A recursive type is a type defined in terms of itself. Recursive types are most often used in defining lists, sequences, or trees of unbounded size. The problem of representing recursive types is similar to the problem of representing recursive function values or any other infinite structure. The basic problem is how to represent the infinite structure in finite space and time while providing convenient mechanisms for manipulating and comparing values of the infinite structure. There are several ways recursive types might be implemented in Phi.

One way to represent recursive types might be to use a circular data structure to represent the type.

Another approach to representing recursive types might be to use abstraction to delay the evaluation of a recursive type.³⁰ Consider the following hypothetical type definition:

```
letrec  $t = (\text{list-of } t)$   
in ...
```

The **list-of** operation is intended to return a list type for any given element type. The hypothetical example above is intended to define a type t that is recursively a list of elements of type t . Some example values of this type might be the empty list $()$, or the list containing two empty lists $((())())$.

Clearly, some kind of delay mechanism is needed to avoid going into an

³⁰Reynolds [Reynolds 85], for example, uses a special **rectype** operation, which is a kind of abstraction mechanism, for expressing recursive types.

infinite loop in trying to evaluate **(list-of *t*)** in the example above.³¹ Function abstraction generally provides a kind of quoting that delays evaluation of the function body until the function is invoked, rather than evaluating the body when the function value (closure) is created.

Now compare the following:

$$\forall t, (\text{element-type-of } (\text{list-of } t)) = t$$
$$\forall t, (\text{apply } (\text{lambda } () t) ()) = t$$

The **list-of** operation creates a list type where the elements must be type *t*, and **element-type-of** returns a list type's element type. Note that these operations deal with *type* values -- they do not create or examine *list* values. **Lambda** (with an empty formal parameter list, in this case) creates a function abstraction, and **apply** applies the function abstraction (to an empty actual parameter list, in this case), as in LISP [McCarthy 66]. The operation **list-of** is analogous to function abstraction, and the operation **element-type-of** is analogous to function application.

The example above showed that there is an analogy between function abstraction and the kind of delay mechanism needed to allow recursive type definitions. Could function abstraction be used to implement recursive types? Certain type operations, such as **list-of**, might act as function abstractions, and one of these would have to enclose each appearance of the

³¹P. Z. Ingerman's *Thunk*, used to implement call-by-name parameter passing in Algol 60, is the classic example of a delay mechanism [Pratt 75]. Lazy evaluation [Henderson 80] is another technique.

type name being recursively defined. (Note that this corresponds to the Algol-68 or Pascal rules for defining recursive types, in which an intervening reference or pointer type must be used in any recursive type definition.) Other type operations, such as `element-type-of`, would act as function application, forcing the element type of the list to be computed, just as function application causes the function body to be evaluated.

This approach has not been worked out for Phi. We do not know if it would be feasible or practical.

7.2.8. ERT Subtypes

One unsatisfactory aspect of ERT triplets $\langle e, r, t \rangle$ is that if the type component is `ert` (indicating that the expression will evaluate to an ERT in the next phase), there is no further information about what type of value might be computed in the following phase. In fact, the Static-IL construct `deep-const` hides the types of constants until the phase before the constant will be used, thus preventing any compiletime type errors regarding that constant from being detected earlier. It would certainly be better to detect all errors as early as possible.

One way to support earlier error detection might be to introduce subtypes of the `ert` type that provide some information about an expression's final type, if known. Consider the following Static-Phi program.

$$\lambda x : t1 \rightarrow t2 . (g x)$$

Recall from Section 3.3.4 that every subexpression starts out being type `ert`; thus the λ expression above will initially be considered type `ert`. But regardless of what types `t1` and `t2` turn out to be, it is syntactically obvious

that the above expression will eventually evaluate to some kind of function value. Hence, it might be useful to initially consider the expression to be a type that is a subtype of **ert**, such as "**ert of fun**", which carries more information than the simple **ert** type carries. Similarly, if **t1** and **t2** happen to be type constants such, as **number**, an even more specific subtype might be returned, such as "**ert of <fun number number>**", which represents the type of an expression that will become a function from **numbers** to **numbers** in some future phase.

We do not know whether ERT subtypes will provide the right practical mechanism for early error detection, or whether some other approach would be better.

7.2.9. Statically Inferred Phases

In Static-Phi, phases are assigned statically by the Translator, based on **emit** and **eval** constructs explicitly embedded in the Static-Phi program. To ease the programmer's burden, it might be possible to have the Phi Translator automatically determine which subcomputations should be performed during which phases, without requiring the programmer to designate them explicitly.

References

- [Ada 82] *Reference Manual for the Ada Programming Language*
Department of Defense, 1982.
- [Balzer 83] Balzer, R., C. Green, and T. Cheatham.
Software Technology In The 1990's: Using A New
Paradigm.
Computer 16(11):39-45, November, 1983.
- [Barendregt 84] Barendregt, H. P.
Studies in Logic and the Foundations of Mathematics.
Volume 103: *The Lambda Calculus: Its Syntax and
Semantics.*
North-Holland, 1984.
- [Beckman 76] Beckman, L., Haraldson, A., Oskarsson, O., and
Sandewall, E.
A Partial Evaluator, and Its Use as a Programming Tool.
Artificial Intelligence 7(4):319-357, Winter, 1976.
- [Boehm 80] Boehm, H., Demers, A., and Donahue, J.
An Informal Description of Russell.
Technical Report TR80-430, Computer Science
Department, Cornell University, 1980.
- [Burstall 84] Burstall, R., and Lampson, B.
A Kernel Language for Abstract Data Types and Modules.
In *Semantics of Data Types, Lecture Notes in Computer
Science*. Proceedings, International Symposium,
Sophia-Antipolis, France, Springer-Verlag, June, 1984.

- [Cheatham 79] Cheatham, T. E., Jr., Holloway, G. H., and Townley, J. A.
Symbolic Evaluation and the Analysis of Programs.
IEEE Transactions on Software Engineering
SE-5(4):402-417, July, 1979.
- [Cheatham 81] Cheatham, T. E., Jr., Holloway, G. H., and Townley, J. A.
Program Refinement by Transformation.
In *The 5th International Conference on Software Engineering*, pages 430-437. Institute of Electrical and Electronics Engineers, Computer Society Press, 1981.
- [Cheatham 84] Cheatham, T. E., Jr.
Reusability Through Program Transformation.
Software Engineering SE-10(5):589-594, September, 1984.
- [Coppo 80] Coppo, M.
An Extended Polymorphic Type System for Applicative Languages.
In *Mathematical Foundations of Computer Science*.
Proceedings of the 9th Symposium Held in Rydzyna, Poland, Springer-Verlag, September, 1980.
- [Demers 79] Demers, A., and Donahue, J.
Revised Report on Russell.
Technical Report TR7-389, Computer Science Department, Cornell University, 1979.
- [Demers 80] Demers, A., and Donahue, J.
The Russell Semantics: An Exercise in Abstract Data Types.
Technical Report TR80-431, Computer Science Department, Cornell University, 1980.
- [Donahue 79] Donahue, J.
On the Semantics of "Data Type".
SIAM Journal of Computing 8(4):546-560, November, 1979.
- [Eggert 81] Paul Richard Eggert.
Detecting Software Errors Before Execution.
Technical Report CSD-810402, UCLA, April, 1981.

- [Ershov 77a] Ershov, A. P.
On the Partial Computation Principle.
Information Processing Letters 6(2):38-41, April, 1977.
- [Ershov 77b] Ershov, A. P., and Grushetsky, V. V.
An Implementation-Oriented Method for Describing
Algorithmic Languages.
In B. Gilchrist (editor), *Information Processing*, pages
117-122. North-Holland, 1977.
- [Ershov 77c] Ershov, A. P., and Itkin, V. E.
Correctness of Mixed Computation in Algol-Like
Programs.
In G. Goos and J. Hartmanis (editors), *Lecture Notes in
Computer Science*, pages 59-77. Springer-Verlag, 1977.
- [Ershov 78] Ershov, A. P.
On the Essence of Compilation.
In E. J. Neuhold (editor), *Formal Description of
Programming Concepts*, pages 391-420. North-Holland,
1978.
- [Ershov 82] Ershov, A. P.
Mixed Computation: Potential Applications and Problems
for Study.
Theoretical Computer Science 18(1):41-67, April, 1982.
- [Flon 82a] Flon, L., and Coopriider, L. W.
*Metaprogramming: Prospects for the Practical Reuse of
Software*.
Technical Report TR-112, Computer Science Department,
University of Southern California, 1982.
- [Flon 82b] Flon, L., and Raeder, G.
Metaprogramming: Language and Examples.
Technical Report TR-113, Computer Science Department,
University of Southern California, 1982.

- [Futamura 71] Futamura, Y.
 Partial Evaluation of Computation Process - An Approach
 to a Compiler-Compiler.
Systems, Computers, Controls 2(5):721-728, September-
 October, 1971.
- [Gordon 79] Goos, G., and Hartmanis, J. (editor).
Lecture Notes in Computer Science. Volume 78:
Edinburgh LCF.
 Springer-Verlag, 1979.
- [Henderson 80] Henderson, P.
Functional Programming: Application and Implementation.
 Prentice-Hall, 1980.
- [Lindsey 71] Lindsey, C. H. and van der Meulen, S. G.
Informal Introduction to Algol 68.
 North-Holland, 1971.
- [Lindsey 83] Lindsey, C. H.
 Elsa: An Extensible Programming System.
 In *IFIP Working Conference on Programming Languages
 and System Design*. IFIP, March, 1983.
- [McCarthy 66] McCarthy, J., Levin, M. et al.
LISP 1.5 Programmer's Manual.
 MIT Press, 1966.
- [Milner 78] Milner, R.
 A Theory of Type Polymorphism.
Journal of Computer and System Sciences 17:348-375,
 1978.
- [Pratt 75] Pratt, T. W.
Programming Languages: Design and Implementation.
 Prentice-Hall, 1984, 1975.

- [Reynolds 85] Reynolds, J. C.
Three Approaches to Type Structure.
In *Mathematical Foundations of Software Development, Proceedings of the International Joint Conference on Theory and Practice of Software Development (TAPSOFT), Volume 1*, pages 97-138. Springer-Verlag, March, 1985.
- [Schooler 84] Schooler, R.
Partial Evaluation as a Means of Language Extensibility.
Technical Report MIT/LCS/TR-324, Laboratory for Computer Science, MIT, August, 1984.
- [Strachey 67] Strachey, C.
Fundamental Concepts in Programming Languages.
Lecture Notes, International Summer School in Computer Programming, Copenhagen.
August, 1967
- [Wegbreit 74] Wegbreit, B.
The Treatment of Data Types in EL1.
Communications of the ACM 17(5):251-264, May, 1974.
- [Wegner 83] Wegner, P.
On the Unification of Data and Program Abstraction in Ada.
In *Principles of Programming Languages, Proceedings of the Tenth Annual Conference*. ACM, Austin, Texas, January, 1983.

Appendix A:

Formal Semantics of Static-Phi and Static-IL

Introduction

This section gives a semantics for phase evaluation of Static-Phi expressions. The semantics of a Static-Phi expression are given by two sets of semantic equations: one set of equations corresponds to translating the Static-Phi expression into a Static-IL expression (in the Expression component of an ERT); the other set corresponds to evaluating a Static-IL expression. This is therefore an operational semantics, though we write it in a denotational style using continuations.

Static-Phi Syntax Domains

$id \in ID$ -- Identifiers.
 $b \in BOOLEAN$ -- Booleans.
 $n \in NUMBER$ -- Numbers.
 $t \in BTYPE$ -- Basic type constants.
 $e \in EXPR$ -- Static-Phi expressions. A program is an EXPR.

Static-Phi Syntax Equations

$ID = \dots$ -- Identifiers.
 $BOOLEAN = \mathbf{true, false}$ -- Booleans.
 $NUMBER = \mathbf{0, 1, 2, \dots}$ -- Numbers.
 $BTYPE = \mathbf{boolean, number, type, ert}$
 -- Basic (non-function) type constants.

```

EXPR = ID           -- Identifier
    + BOOLEAN      -- Boolean constant
    + NUMBER       -- Number constant
    + BTYPE        -- Basic (non-function) type constant
    + ( funtype EXPR EXPR )
                    -- For expressing the types of functions
    + ( emit EXPR ) -- Normal runtime phase is one phase later
    + ( eval EXPR ) -- Normal runtime phase is one phase
earlier
    +  $\lambda$  ID : EXPR  $\rightarrow$  EXPR . EXPR
                    -- Abstraction, with parameter, return types
    + ( EXPR EXPR ) -- Function application

```

Static-IL (Semantic) Domains

These domains are best interpreted as semantic domains, though they are sometimes used as though they were syntactic domains. The reason for this is to avoid having to deal with the cumbersome detail of two parallel domains -- one syntactic and one semantic -- and a trivial semantic correspondence between them.

```

id  $\in$  ID           -- Identifiers. Same domain as in Static-Phi.

b  $\in$  BOOLEAN      -- Booleans. Same domain as in Static-Phi.

m,n  $\in$  NUMBER     -- Numbers. Same domain as in Static-Phi.

t  $\in$  TYPE         -- Types. Includes both basic types and function types.

r  $\in$  RENV         -- Required-environment. Lists identifiers and their types.

e  $\in$  EXPR        -- Static-IL expressions. A program is an EXPR.

ert  $\in$  ERT        -- Triplet  $\langle e, r, t \rangle$ : e is an expression, r is a list of
                    identifier-type pairs, and t is a type. We deal only with a
                    restricted set of ERT triplets, for which r lists all free
                    variables and their types in e, and e is guaranteed to
                    evaluate to a value of type t (or some error condition).

```

$env \in ENV$ -- Environments
 $v \in EV$ -- Expressible values.
 $k \in ECONT$ -- Expression continuations.
 $err \in ERROR$ -- "Compiletime" errors of various kinds.

Static-IL Domain Equations

$ID = ID$ -- Identifiers from Static-IL.
 $BOOLEAN = BOOLEAN$ -- Booleans from Static-IL.
 $NUMBER = NUMBER$ -- Numbers from Static-IL.
 $FTYPE = \{fun\} \times TYPE \times TYPE$ -- Function type: domain, range.
 $TYPE = BTYPE$ -- Basic types,
 + $FTYPE$ -- Function types.
 $RENV = \{\langle \rangle\}$ -- Required-environment.
 + $(ID \times TYPE) \times RENV$ (Identifier, type pairs.) Note that
 required-environments are represented
 slightly differently in this appendix than
 in
 the body of this work.
 $EXPR = ID$ -- Static-IL expressions.
 + **(quote EV)**
 + **(incr EXPR)**
 + **(check-funtype EXPR EXPR NUMBER)**
 + **(funtype EXPR EXPR)**
 + **(check-check-lambda ID EXPR EXPR EXPR NUMBER)**
 + **(check-lambda ID EXPR EXPR EXPR)**
 + **(lambda ID EXPR)**
 + **(check-apply EXPR EXPR)**
 + **(apply EXPR EXPR)**
 + **(deep-const EV TYPE NUMBER)**

ERT = EXPR x RENV x TYPE -- ERT triplet. Free variables of EXPR are listed in RENV with their types. EXPR evaluates to a value of type TYPE.

CLOSURE = ID x EXPR x ENV-- Function closures

EV = BOOLEAN -- Expressible values
+ NUMBER
+ TYPE
+ ERT
+ CLOSURE

ENV = ID → EV -- An environment is a function from identifiers to expressible values. Note environments are represented slightly differently in this appendix than in the body of this work.
that

ECONT = EV → [EV + ERROR] -- An expression continuation.

ERROR = { **error-non-type,** -- "Compiletime" errors possible.
error-non-function,
error-inconsistent-req-envs,
error-type-mismatch,
error-different-type-used-in-body,
error-ert-expected,
error-non-ert,
error-body-is-not-ert,
error-body-and-range-types-differ,
error-arg-ready-before-function }

Meta-Language Notation

The translation rules and semantic equations will use a meta-language including **if**, **where**, **let** and **maximum** constructs. They are written in **this font**. Comments on a line are preceded by "--". Tuples are written, for example, as "<a, b>". Function application is written, for example, as "(f x)". The continuation-style operator ";" also denotes function application,

but it is right associative and binds weakly. Thus, "f; g; x" means "(f (g x))". The body of a lambda abstraction " $\lambda x . \dots$ " extends as far to the right as possible.

We use the operators "+" and "-" to concatenate two Required-environments, and to remove all occurrences of an identifier from a Required-environment. We use the notation $\text{env}[v/id]$ to denote the environment env augmented by the binding of v to id . (Remember that a "Required-environment" is not the same as an "environment"!) More formally, we can recursively define these operations:

$$r1 + r2 \hat{=} \begin{array}{l} \text{if } r1 = \langle \rangle \\ \text{then } r2 \\ \text{else let } \langle \langle id1, t1 \rangle, r1' \rangle = r1 \\ \text{in } \langle \langle id1, t1 \rangle, r1' + r2 \rangle \end{array}$$

$$r - id \hat{=} \begin{array}{l} \text{if } r = \langle \rangle \text{ then } r \\ \text{else let } \langle \langle id1, t1 \rangle, r' \rangle = r \\ \text{in } \begin{array}{l} \text{if } id1 = id \text{ then } r' - id \\ \text{else } \langle \langle id1, t1 \rangle, r' - id \rangle \end{array} \end{array}$$

$$\text{env}[v/id] \hat{=} \lambda id1 . \text{if } id1 = id \text{ then } v \text{ else } \text{env}(id1)$$

(Note that the equality "=" used here between identifiers is true iff the two identifiers are same identifier -- it has nothing to do with the values of those identifiers.)

Translating from Static-Phi to Static-IL

A Static-Phi expression is not evaluated directly. Instead, it is first translated to a corresponding Static-IL expression (contained in an ERT), which is in turn evaluated through one or more phases. The function *Trans-count* is

applied to the Static-Phi program and produces the Static-IL translation by calling auxiliary functions *Trans* and *Count*. These functions have the following types:

Trans-count: EXPR → ERT

Count: EXPR X NUMBER → NUMBER

Trans: EXPR X NUMBER → ERT

The TYPE component of the ERT that *Trans* or *Trans-count* returns will always be **ert**, the EXPR component will be the Static-IL expression corresponding to EXPR, and the RENV component will list all the free variables appearing in that EXPR component. Each variable is initially type **ert**. *Trans-count* is simply defined as follows:

$$\text{Trans-count}[e] = \text{Trans}[e, \text{Count}[e, 0]]$$

Function *Count* is used to count the depth of the minimum number of phases required, and *Trans* does the real translation work. These functions are defined below.

Auxiliary Function "Count"

Function *Count* actually counts a depth, which may be positive or negative, rather than the number of phases required. The parameter *n* represents the current normal runtime phase -- the number of phases before some arbitrary phase 0. Thus, a more positive *n* indicates an earlier phase, and a less positive (or negative) *n* indicates a later phase. Thus, this numbering is the

opposite from phase numbering used in previous chapters of this work. The reason for this is that in translation and phase evaluation the emphasis is on the number of phases required, rather than the number of phases that have already been performed. The following rules define *Count*.

$$\text{Count}[\text{id}, n] = n$$

An identifier does not need any extra type checking phases.

$$\text{Count}[\text{b}, n] = n+1$$

$$\text{Count}[\text{m}, n] = n+1$$

$$\text{Count}[\text{t}, n] = n+1$$

Constants need only one extra phase for type checking.

$$\begin{aligned} \text{Count}[(\text{funtype } e1 \ e2), n] = \\ \text{Maximum}\{ n+1, \text{Count}[e1,n], \text{Count}[e2,n] \} \end{aligned}$$

The *funtype* construct itself needs one extra phase for type checking, but the subexpressions may need more, so we take the maximum.

$$\begin{aligned} \text{Count}[\lambda \text{id} : e1 \rightarrow e2 . e3, n] = \\ \text{Maximum}\{ n+2, \text{Count}[e1,n+1], \text{Count}[e2,n+1], \text{Count}[e3,n] \} \end{aligned}$$

The λ construct requires two extra phases: one to type check the function itself, and one to check the types of the domain and return type expressions. Of course, the subexpressions may need more, so, as with *funtype*, we take the maximum. Also note that subexpressions *e1* and *e2* are implicitly inside an *eval*; hence the "n+1"s.

$$\begin{aligned} \text{Count}[(e1 \ e2), n] = \\ \text{Maximum}\{ n+1, \text{Count}[e1,n], \text{Count}[e2,n] \} \end{aligned}$$

Function application itself needs one extra phase for type checking, but the subexpressions may need more, so again we take the maximum. --

$$\text{Count}[\text{emit } e, n] = \text{Count}[e, n-1]$$

The `emit` construct is not a runtime notion at all. The number of phases required just depends on the subexpression, but note that its normal runtime phase will one phase later. *Trans* will take that into account during translation, so we anticipate it here by subtracting one from the current depth.

$$\text{Count}[\text{eval } e, n] = \text{Count}[e, n+1]$$

The inverse of `emit`.

Translation Rules

$$\text{Trans}[\text{id}, n] = \langle \text{id}, \langle \langle \text{id}, \mathbf{ert} \rangle, \langle \rangle \rangle, \mathbf{ert} \rangle \text{ -- Identifiers}$$

Each identifier is initially type **ert**. The EXPR component is simply the identifier, hence the required-environment only lists this one identifier of type **ert** as the free variables appearing in it.

$$\begin{aligned} \text{Trans}[b, n] &= \langle e, \langle \rangle, \mathbf{ert} \rangle, && \text{-- Boolean constants} \\ &\text{where } e \in \text{EXPR} = (\mathbf{deep-const } b \text{ boolean } n-1) \end{aligned}$$

A constant is translated to an ERT in which the EXPR component is a **deep-const** expression. There are no free variables in it; hence the required-environment component of the returned ERT is empty.

$$\begin{aligned} \text{Trans}[m, n] &= \langle e, \langle \rangle, \mathbf{ert} \rangle, && \text{-- Number constants} \\ &\text{where } e \in \text{EXPR} = (\mathbf{deep-const } m \text{ number } n-1) \end{aligned}$$

Similar to boolean constants.

Trans[*t*, *n*] = < *e*, <>, **ert** >, -- Basic type constants
 where *e* ∈ EXPR = (**deep-const** *t* **type** *n*-1)

Similar to boolean constants.

Trans[(funtype *e1 e2*), *n*] = -- Types of functions
 let < *e1'*, *r1*, **ert** > ∈ ERT = *Trans*[*e1*, *n*],
 < *e2'*, *r2*, **ert** > ∈ ERT = *Trans*[*e2*, *n*]
 in < (**check-funtype** *e1' e2' n*-1), (*r1* + *r2*), **ert** >

The subexpressions are translated to ERT's, and their RENV (required-environment) and EXPR components are combined to form the resulting ERT. The expression components simply become the subexpressions of a **check-funtype** expression -- they will evaluate to ERTs in the first evaluation phase. The required-environments are simply concatenated because the free variables of the whole **check-funtype** expression are simply the free variables of the subexpressions *e1'* and *e2'*. All variables are type **ert** in the first phase.

Trans[λ *id* : *e1* → *e2* . *e3*, *n*] = -- Abstraction
 let < *e1'*, *r1*, **ert** > ∈ ERT = *Trans*[*e1*, *n*-1],
 < *e2'*, *r2*, **ert** > ∈ ERT = *Trans*[*e2*, *n*-1],
 < *e3'*, *r3*, **ert** > ∈ ERT = *Trans*[*e3*, *n*]
 in <
 (**check-check-lambda** *id e1' e2' e3' n*-2),
 (*r1* +(*r2* + (*r3* - *id*))),
 ert
 >

The subexpressions are translated to ERTs, then combined to form the resulting ERT. Subexpressions *e1* and *e2*, which give the function's domain and range types, are inside an implied **eval**; hence they are translated so that the function's domain and range types will be computed one phase before the function value (closure) is computed. The EXPR component of the resulting

ERT simply uses the original bound variable, *id*, and the EXPR components from translating the subexpressions to form the **check-check-lambda** expression. The required-environments are combined, but since *id* is a locally bound variable inside the body expression *e3'*, it is removed from *r3* before being combined with *r1* and *r2*. However, *id* is not locally bound in *e1'* or *e2'* (i.e. *e1'* and *e2'* are in an outer scope) so it is not removed from *r1* or *r2*.

```

Trans[ ( e1 e2 ), n ] = -- Function application
  let < e1', r1, ert > ∈ ERT = Trans[ e1 , n ],
      < e2', r2, ert > ∈ ERT = Trans[ e2 , n ]
  in < (check-apply e1' e2' ), (r1 + r2), ert >

```

The subexpressions are translated and combined to form the resulting **check-apply** ERT.

Phase Evaluation

We now define a function, *Pheval*, that evaluates a Static-IL expression relative to some environment *env*. *Pheval* has the following type:

Pheval: EXPR → ENV → ECONT → [EV + ERROR], or equivalently:

Pheval: EXPR → ENV → [EV → [EV + ERROR]] → [EV + ERROR].

The environment *env* is a function, with the following type:

env : ENV = ID → EV

Pheval uses two auxiliary functions: *Funtype?* and *Consistent?*. *Funtype?* is used

on TYPEs. It is **true** for functional types, i.e. types of the form $\langle \text{fun } t1, t2 \rangle$, for some types $t1$ and $t2$. It is not defined here, but has the following type: --

Funtype?: TYPE \rightarrow BOOLEAN

Auxiliary function *Consistent?* checks whether the types of identifiers in two required-environments are consistent. In other words, for each identifier and type $\langle \text{id}, t \rangle$ in the first required-environment, *Consistent?* checks every identifier-type pair $\langle \text{id}', t' \rangle$ in the second required-environment, and returns **false** if two identifiers id and id' match but their types t and t' differ. Otherwise it returns **true**. *Consistent?* has the following type:

Consistent?: RENV \times RENV \rightarrow BOOLEAN

Formally, *Consistent?* can be recursively defined as follows.

$$\begin{aligned} \text{Consistent?}(r1, r2) &\hat{=} \\ &\text{if } r1 = \langle \rangle \text{ or } r2 = \langle \rangle \\ &\text{then true} \\ &\text{else let } \langle \langle \text{id1}, t1 \rangle, r1' \rangle \in \text{RENV} = r1, \\ &\quad \langle \langle \text{id2}, t2 \rangle, r2' \rangle \in \text{RENV} = r2 \\ &\text{in if } \text{id1} = \text{id2} \text{ and } t1 \neq t2 \\ &\quad \text{then false} \\ &\quad \text{else Consistent?}(r1', r2) \text{ and Consistent?}(r1, r2') \end{aligned}$$

Phase Evaluation Semantic Rules

Pheval[**id**](*env*)(*k*) = *k*(*env*(*id*)) -- Identifier

The value of the identifier is simply retrieved from the environment. For at least the first phase after translation, the identifier is guaranteed to evaluate to an ERT. In a subsequent phase, it may evaluate to a value of some other type.

Pheval[(**quote** *v*)](*env*)(*k*) = *k*(*v*) -- Quoted value

A quoted value is simply returned as is. Quoted values may be of various types.

Pheval[(**incr** *e*)](*env*)(*k*) = -- Increment (add 1)
 Pheval[*e*](*env*);
 $\lambda v \in EV . k(v+1)$

The subexpression is evaluated (it will be a number), and the resulting value plus one is passed on to the continuation.

Pheval[(**check-funtype** *e1 e2 n*)](*env*)(*k*) = --For function type
 Pheval[*e1*](*env*);
 $\lambda \langle e1', r1, t1 \rangle \in ERT . Pheval[e2](env);$
 $\lambda \langle e2', t2, t2 \rangle \in ERT .$
 if *Consistent?*(*r1*, *r2*)
 then if *n* > 0
 then if *t1* = *ert* and *t2* = *ert*
 then *k*(\langle (**check-funtype** *e1' e2' n-1*), *r1+r2*, *ert* \rangle)
 else **error-ert-expected** -- Needed an ERT.
 else if *t1* = *type* and *t2* = *type*
 then *k*(\langle (**funtype** *e1' e2'*), *r1+r2*, *type* \rangle)
 else **error-non-type** -- Needed *type*.
 else **error-inconsistent-req-envs**

Excluding errors, **check-funtype** always evaluates to an ERT, passing it on to the expression continuation. The purpose of **check-funtype** is to generate a **funtype** expression whose subexpressions are guaranteed to evaluate to

TYPE's. In contrast with **funtype**, the subexpressions of **check-funtype** evaluate to ERTs.

Subexpressions **e1** and **e2** are first evaluated; they evaluate to intermediate ERTs. These intermediate ERTs will be combined to form the resulting ERT, whose expression component will either be a **check-funtype** or a **funtype** EXPR. The required-environment components of the intermediate ERTs must be consistent, since they will be concatenated to form the required-environment of the resulting ERT. The phase depth *n* determines whether a **check-funtype** or a **funtype** expression is to be generated. If *n*>0, we still have one or more phases to go before we should generate a **check-funtype** expression, so the subexpressions must evaluate to ERTs; otherwise (when *n*=0), we must generate a **funtype** expression, and its subexpressions must evaluate to TYPEs.

Pheval[**(funtype e1 e2)**](env)(k) = --Function type
 Pheval[**e1**](env);
 λ t1 ∈ TYPE . *Pheval*[**e2**](env);
 λ t2 ∈ TYPE . k(<**fun** t1, t2 >) -- t1 is domain; t2 is range.

Excluding errors, **funtype** always evaluates to a TYPE. In contrast with **check-funtype**, **funtype**'s subexpressions both evaluate to TYPEs. The final result will be a function type, containing the types of the function's domain and range, obtained from evaluating subexpressions **e1** and **e2**.


```

Pheval[ (check-check-lambda id e1 e2 e3 n ) ](env)(k) =
  Pheval[ e1 ](env);
  λ <e1', r1, t1> ∈ ERT . Pheval[ e2 ](env);
  λ <e2', r2, t2> ∈ ERT . Pheval[ e3 ](env[<id,<<id,ert>,<>>,ert>/id]);
  λ <e3', r3, t3> ∈ ERT .
    if Consistent?( r1, r2 )
    then if Consistent?(<<id,ert>,<>>,r3) and Consistent?(r1+r2,(r3-id))
    then if t3 = ert
    then if n > 0
    then if t1 = ert and t2 = ert
    then let e=(check-check-lambda id e1'
      e2' e3' n-1)
      in k( < e, (r1+r2+(r3-id)), ert > )
    else error-non-ert -- Needed ERT
    else if t1 = type and t2 = type
    then let e=(check-lambda id e1' e2' e3' )
      in k( < e, (r1+r2+(r3-id)), ert > )
    else else error-non-type
      -- Not a type expr
    else error-body-is-not-ert -- Body should be ERT
    else error-different-type-used-in-body
      -- Clash of used/expected types
    else error-inconsistent-req-envs

```

Excluding errors, **check-check-lambda** always evaluates to an ERT and passes it on to the expression continuation. The purpose of **check-check-lambda** is to generate a **check-lambda** whose first two subexpressions are guaranteed to evaluate to values of type TYPE. In contrast to **check-lambda**, all of **check-check-lambda**'s subexpressions evaluate to ERT's.

Bound variable **id** is local to subexpression **e3** (**e3** is in a new scope), and will be bound to an ERT within **e3**, whereas subexpressions **e1** and **e2** are considered to be in some outer scope. We first evaluate **e1** and **e2** in the outer environment, and then evaluate **e3** in an environment in which the

bound variable *id* is bound to the following ERT: $\langle id, \langle \langle id, \mathbf{ert} \rangle, \langle \rangle \rangle, \mathbf{ert} \rangle$. The expression component is simply the identifier, and it will evaluate to an ERT; hence the type component is **ert**, and the only free variable listed in the required-environment component is the identifier itself. In effect, this declares instances of *id* already appearing in the body to be type ERT. (New instances may be introduced, however, when subexpressions evaluate to ERTs, as with macro expansion.)

The first *Consistent?* test ensures that any variables used in the domain and range subexpressions are the same types. The second and third *Consistent?* tests ensure that the body expects the formal parameter to be type **ert** and that any other free variables appearing in the body have the same types as they do in the domain and range subexpressions. These tests are necessary because subexpressions that evaluate to ERT's can introduce new references to bound variables.

If the phase depth $n > 0$, we have to generate another **check-check-lambda**, in which case the domain and range subexpressions must again evaluate to ERTs; otherwise, we generate a **check-lambda** and the domain and range subexpressions must evaluate to TYPEs.

```

Pheval[ (check-lambda id e1 e2 e3 ) ](env)(k) = -- e1, e2 will be TYPEs
  Pheval[ e1 ](env);
  λ t1 ∈ TYPE . Pheval[ e2 ](env);
  λ t2 ∈ TYPE . Pheval[ e3 ](env[ <t1,<<id,t1>,<>>,id> / id ]);
  λ < t3, r3, e3' > ∈ ERT .
    if Consistent( <<id,t1>,<>>,id>, r3 )
    then if t2 = t3
      then k( < (lambda id . e3' ), r3 - id, <fun t1 t2 > > )
      else error-body-and-range-types-differ
    else error-different-type-used-in-body
      -- id has different type in body

```

Excluding errors, **check-lambda** always evaluates to an ERT. Its purpose is to generate a **lambda** expression whose body expects the formal parameter to be the type declared for it.

Type subexpressions **e1** and **e2** are evaluated to types **t1** and **t2**, then body subexpression **e3** is evaluated to an ERT in an environment that includes a binding of the formal parameter **id** to the ERT $\langle t, \langle \langle id, t \rangle, \langle \rangle \rangle, id \rangle$. In effect, this declares existing instances of **id** in the body to be type **t**. The *Consistent?* test is used to verify that a new instance of the formal parameter with a different type has not been injected into the body expression (as can happen with macro expansion). Finally, the body type must agree with the function's declared range type.

```

Pheval[ (lambda id . e ) ](env)(k) = k( < id, e, env > ) -- Create a
closure.

```

This is a function abstraction. To implement lexical scoping, a closure of the bound variable, expression body, and current environment is created and returned.

```

Pheval[ (check-apply e1 e2 ) ](env)(k) =  -- e1 and e2 will be ert's
  Pheval[ e1 ](env);
  λ < e1', r1, t1 > ∈ ERT . Pheval[ e2 ](env);
  λ < e2', r2, t2 > ∈ ERT .
    if Consistent?( r1, r2 )
    then if Funtype?( t1 )
      then let <fun t11, t12 > ∈ FTYPE = t1 -- Domain, range
        in   if t11 = t2
          then k( < (apply e1' e2' ), r1+r2, t12 > )
          else error-type-mismatch
              -- Formal-actual type mismatch
        else if t1 = ert
          then if t2 = ert
            then k( < (check-apply e1' e2' ), r1+r2, ert> )
            else error-arg-ready-before-function
          else error-non-function  -- Not function or ert
      else error-inconsistent-req-envs

```

Excluding errors, **check-apply** will always evaluate to an ERT. Its purpose is to generate an **apply** expression that has been type checked to guarantee that the first argument will evaluate to a function, and the second argument will evaluate to the type declared for the function's formal parameter. In contrast with **apply**, the subexpressions of **check-apply** both evaluate to ERTs.

Subexpressions **e1** and **e2** are evaluated to ERTs, and the required-environments of these ERTs must be consistent; they are checked as in previous cases. If **t1** is a function type, the function subexpression **e1'** will evaluate to a function to be applied to the actual parameter in the next phase; hence the type of the actual parameter must match the function's declared formal parameter type, and an **apply** ERT will be generated. Otherwise, **t1** should be **ert**, indicating that the function subexpression will again evaluate to an ERT during the next phase. In this case, **t2** should also be **ert** (indicating that the argument subexpression will also evaluate to an ERT), and another

check-apply will be generated. At this point, it is an error if **t2** isn't **ert**, since this means that the argument is ready to evaluate to some fixed, non-ERT value before the function expression is ready to evaluate to a function value.

$$\begin{aligned}
 Pheval[\text{(apply } e1 \ e2 \) \](env)(k) &= && \text{--Function application} \\
 Pheval[e1 \](env); & \\
 \lambda \langle id, e, env' \rangle \in \text{CLOSURE} . Pheval[e2 \](env); & \\
 \lambda v \in EV . Pheval[e \](env'[v/id])(k) &
 \end{aligned}$$

Normal function application. Subexpression **e1** is guaranteed to evaluate to a function closure, and **e2** evaluates to the actual parameter, the type of which is guaranteed to be the domain type of the function.

Proving That No Runtime Type Errors Are Possible

This section briefly briefly sketches how to approach proving the assertion that runtime type errors are not possible in Static-IL. The more specific assertion is that every ERT generated by this system is *valid*. (This will be clarified below.) Overall, the proof is by induction on the number of phases used to produce the ERT. The basis is zero phases, when the ERT is produced directly by the Translator. Both the base case and the induction step are, in turn, proved using structural induction on the original Static-Phi program or the Static-IL Expression component of the ERT.

First off, we must define what we mean by "runtime type error" in order to show that such errors are not possible. The easiest way to do this is probably to add an explicit type tag to the values that are manipulated by Static-IL programs, and then to define runtime type errors in terms of these type tags.

Next we must specifically define what it means for an environment to *satisfy* a Required-environment. The environment must supply bindings of the proper types for all of the identifiers listed in the Required-environment.

Now, we really want to prove that every ERT produced by this system is "valid", so we must define "valid". Basically, an ERT $\langle e, r, t \rangle$ is *valid* if, in an environment that satisfies the Required-environment r , e is guaranteed to evaluate to a value of type t (or to some compile-time error value) without incurring any runtime type errors.

With the proper definitions in order, the proof would proceed by induction on the number of phases used to produce the ERT.

Basis. The basis is when zero phases were used to produce the ERT, that is, we must first show that the translator always produces a valid ERT. This part would be done using structural induction on the original Static-Phi program. The most important thing to note in this part is that the result of the *Count* function used in translation is completely irrelevant to the proof. The n parameter used by several of the Static-IL constructs to determine how many phases to wait, has no bearing on the type correctness of the system.

Inductive hypotheses. Next, we consider any valid ERT $\langle e, r, t \rangle$, and any environment env that satisfies the required environment r . Thus, the basic inductive hypothesis is that $\langle e, r, t \rangle$ is valid and that the environment env satisfies the Required-environment r . But furthermore, we must construct the right hypothesis on the environment to ensure that no Trojan horse runtime type errors can sneak in through the environment. Every ERT value that

comes from the environment must be valid, and every function that comes from the environment must be assured to execute without runtime type errors.

Induction. We must now prove that if $t = \mathbf{ert}$ then $Pheval[\mathbf{e}](env)(\lambda v.v)$ is either a valid ERT or one of the compiletime error values. This would proceed by structural induction on \mathbf{e} .

