Functional Encapsulation and Type Reconstruction in a Strongly-typed, Polymorphic Language

Shail Aditya Gupta

MIT / LCS / TR-647
February 1995

This report describes research done at the Laboratory of Computer Science of the Massachusetts Institute of Technology. Funding for this work has been provided in part by the Advanced Research Projects Agency of the Department of Defense under the Office of Naval Research contract N00014-92-J-1310.

This report was originally published as the author's doctoral dissertation.

Functional Encapsulation and Type Reconstruction in a Strongly-typed, Polymorphic Language

Shail Aditya Gupta

Technical Report MIT / LCS / TR-647
February 1995

MIT Laboratory for Computer Science
545 Technology Square
Cambridge MA 02139

Abstract

Static type systems are traditionally used to prevent run-time type-errors in user programs and to assign appropriate storage representations to objects during compilation. In this thesis, we explore some new ways of using static type information in the design, compilation, and execution of programs written in a strongly-typed, polymorphic language.

Programmers often find it useful to know whether or not a particular data-structure may be updated outside a given control block. Information about an object's non-mutability helps compiler optimizations, improves aliasing and dependence analyses, and permits unrestricted caching of functional data at run-time. In the first part of this thesis, we present a safe, static mechanism for *functional encapsulation* of imperative data-structures using a powerful type system based on closure types and regions. We introduce a new language construct called `close` which delimits the scope of side-effects on imperative objects and converts them into functional objects outside that scope. This mechanism may be used to build efficient, high-level, functional data-abstractions within a language using its low-level, imperative constructs. Type-safety and non-mutability of closed objects is guaranteed by a semantic soundness theorem that ensures consistency between the static and the dynamic semantics. The type system is presented in the context of Id, which is a strongly-typed, polymorphic, higher-order language, and it easily simplifies to a first-order, monomorphic language such as C or Fortran.

In the second part of the thesis, we develop a general, compiler-directed methodology for complete *type reconstruction* of run-time objects in a polymorphic language without using any run-time type-tags. Run-time type reconstruction is carried out by instantiating static type information for each function activation frame present within the dynamic call tree. Additional *type-hints* are inserted automatically at compile-time and are decoded at run-time to ensure complete type reconstruction. We present the necessary compiler analysis and the type reconstruction algorithm and prove their correctness. This technique has been used successfully for displaying run-time objects within the Id source debugger for Monsoon and to perform tagless garbage collection in the *T architecture. We describe the latter application in detail, comparing its performance with other schemes for automatic storage reclamation.

Key Words and Phrases: Functional Encapsulation, Operational Semantics, Polymorphism, Type Soundness, Imperative Typing, Closure Typing, Regions, Typed Run-time System, Type Reconstruction, Type Conservation, Type-Hints, Tagless Garbage Collection.

Acknowledgments[†]

I am most grateful to my thesis advisor and my mentor, Prof. Arvind, without whose guidance and encouragement I would not have seen this day. He showed a keen interest in me when I first joined MIT in the fall of 1987 and took his course on Dataflow and Reduction Architectures (6.847). I was highly impressed by his magnetic personality, a clear vision of the future of parallel computing, and a strong conviction for achieving that research goal. After seven years, I still remain deeply impressed, and in many ways, greatly influenced by his personality and research ideas. I thank him for giving me the opportunity to work with him and with the other members of the Computation Structures Group, which for the last seven years, has been a truly exciting and friendly research atmosphere to work in.

I would also like to thank the other members of my thesis committee, Prof. Albert R. Meyer and Dr. Rishiyur S. Nikhil, who provided valuable advice and guidance from time to time and helped me get through my thesis defense with ease. I am especially grateful to Dr. Nikhil who previously supervised my Master's thesis. He helped me shape my graduate academic career and has been a continuous source of support and inspiration.

Xavier Leroy from INRIA, and Satyan Coorg from CSG gave enormous technical help while developing the type system presented in the first part of this thesis. Prof. Arvind provided valuable insights into the language design issues that led to the current design of the `close` construct. James Hicks from MCRC and Christine Flood from CSG helped in implementing and comparing various storage reclamation schemes described in Chapter 8. I sincerely thank them all for their time and patience in helping me complete this research.

I heartily thank all the members of the Computation Structures Group, both past and present, for their continuing friendship and support, making it all feel like a big family. I have made some of the best friends of my life in this group both professionally and personally. I would especially like to thank Zena Ariola for providing moral support and sound advice, and Michael Halbherr for his delightful friendship and a wonderful time in Europe.

I would also like to thank Gita, Prof. Arvind's wife, whose love and care for me, home cooked food, and invitations to participate in family festivities really provided me with the feeling of a "home away from home".

I also thank all my other friends at MIT and elsewhere, and members of the music group *Gunjan* for their enjoyable company and memorable experiences, creating welcome diversions from work and making these past several years some of the most cherished moments of my life.

I am eternally grateful for the love and affection bestowed upon me from my family. The encouragement and support I received from my parents, Vidyaratna and Kusum, is beyond measure. This thesis is as much a fulfillment of their dream as it is of mine. It is the fruit of their tremendous confidence in my abilities under all circumstances. I am very grateful to my brothers, Vikram and Uday, for their love and support and who took all my responsibilities at home upon themselves. I am also extremely grateful to my sisters, Archana and Kalpana, who provided me with enormous love and affection as well as strong moral support and sound advice during difficult times.

Finally, I am thankful beyond words to The Almighty God for keeping me steady on my path, giving me the strength and will power to do the right thing every time, helping me to be at peace with myself in the face of sorrow or joy, and ultimately making my lifelong academic dream come true. May He continue to guide my path in the same way in future. Amen.

[†]Funding for this work has been provided in part by the Advanced Research Projects Agency of the Department of Defense under the Office of Naval Research contract N00014-92-J-1310.

To my parents,
Vidyaratna and Kusum.

Contents

List of Figures

Chapter 1

Introduction

One of the main goals of modern, high-level programming languages is to provide an intuitive programming model that is useful in writing applications and reasoning about their behavior. Some languages enforce a style of programming that guarantees useful properties for the programs written in those languages. In this thesis, we concentrate on a class of high-level languages that are *strongly-typed*. Strong-typing enforces *type-consistency* which imparts a degree of robustness to the program. A type-consistent program is guaranteed never to run into a run-time type-error, *e.g.*, attempting to use an integer in a floating-point computation or applying a non-function object to an argument.

In a strongly-typed language, type-consistency can be enforced during compilation (*static typing*) or during execution (*dynamic typing*). The compiler for a statically-typed language has to be somewhat conservative in enforcing type-consistency: it may reject certain programs that appear to be inconsistent, although such programs may not encounter a run-time type-error for certain inputs (or even for all possible inputs). The advantage of being conservative is that a program that has been statically determined to be type-consistent, is guaranteed to execute in a type-consistent manner for all inputs. Therefore, no checks for type-consistency need to be made during its execution.

Type information is primarily used in statically-typed languages to check for type-consistency within programs and to choose memory representations for data-structures. Most of this information is thrown away once a program has been compiled. At most, some type information may be saved in a symbol table to be used by a source-level debugger. In this thesis, we wish to explore more fundamental ways in which to incorporate and use type information in the design, compilation, and execution of a program written in a strongly-typed language. We wish to use the type system of our language as a tool for structuring the language design into tight abstraction layers, provide support for compiler optimizations and automatic code generation as well as support for run-time facilities such as source-level debugging and garbage collection.

1.1 Layered Language Design

Modern, high-level languages offer a variety of data and control abstraction mechanisms to enable users to structure their applications properly. Most programming language designs fall into one of the following two categories: either a language includes a large repertoire of common datatypes and their manipulation functions as part of its definition as in Common Lisp [SJ90], or these objects are defined separately in a standard prelude or in system and user libraries as in the case of Standard ML [MT91, MTH90], Haskell [HWe90], or C [Pla92]. The first approach sometimes leads to language definitions that may be too large to understand, implement and reason about. The second approach usually leads to small and simple *language kernels* that may be used to "implement" high-level datatypes and their associated functions as independent libraries. This approach seems better in terms of overall ease of understanding and maintainability of the language, though it requires a careful design of the libraries and their user interface.

The recent success of strongly-typed, polymorphic, functional languages, such as Haskell and Standard ML, highlight the importance of this *layered* approach to language design. Small language kernels have manageable semantic complexity and can be subjected to powerful reasoning techniques. At the same time, a small set of kernel primitives that can be suitably mapped to the underlying architecture provide a flexible and efficient means of implementing pre-defined and user-defined high-level datatypes. In order for such kernel implementations to be sound and transparent to the end-user, a proper data and type abstraction mechanism must be provided in the kernel language. Otherwise, the semantic correctness of the implementation may be in doubt. An example of this situation is the C language [KR88] which offers complete flexibility of a low-level kernel language but lacks a tight abstraction mechanism, leading sometimes to subtle errors in user programs. For this reason, many high-level languages offer only a fixed set of high-level constructs with pre-defined semantics rather than provide the user with the complete flexibility and the raw power of a low-level kernel language. The list comprehension mechanism, first introduced in NPL [Bur77, Dar77] and later adopted in Miranda [Tur85] and Haskell, is an example of such a language construct.

From a language design standpoint, a powerful type system can be used to enforce the type abstraction desired for kernel language implementations of high-level datatypes in libraries without changing the high-level language definition or modifying the compiler. In Part I of this thesis, we are going to present a type system that will allow us to build a data-structure in a low-level, imperative style and then safely *encapsulate* it as a functional data-structure. The motivation for doing so is as follows.

First, our type system reduces the complexity of writing compilers for functional languages. Functional syntactic constructs, such as list and array comprehensions, that have to be implemented within the compiler as primitive constructs, can instead be desugared into ordinary functions that are implemented in an independent system library. This is possible because we allow the programmer to use low-level imperative constructs while implementing the library that are safely encapsulated within functional abstractions provided by the type system. This approach is also very flexible since it allows modification and extension of existing language constructs as well as addition of new constructs without disturbing the bulk of the compiler.

Second, our type system provides a way to safely implement functional computations using imperative algorithms that cannot otherwise be expressed in a functional style efficiently. Notable examples that have this characteristic are accumulation (histogramming) algorithms and graph algorithms. Although, the final result may be functional, the computation often needs to be performed in an imperative way in order to achieve efficiency in space and time. Using our type system, an imperative computation can be safely embedded within a functional program while still preserving its clean semantics and simple reasoning.

From the standpoint of a compiler, working hand-in-hand with a powerful type system can prove to be more fruitful than working around it, as most compilers tend to do. Static types of program fragments provide valuable information about "what" is being computed. The shape and size of data-structures and the input/output parameters of functions can be determined using their static types. Intelligent compilers can use this information while performing important optimizations such as boxing/unboxing of data, code specialization, and register allocation.

Unfortunately, very few compilers actually propagate the full source type information all the way to the back-end, the Glasgow Haskell compiler [PJ92] being a notable exception. In a layered language, this task is considerably simplified since only a small number of kernel language constructs are involved within the later phases of the compiler.

It is also possible to use source type information at run-time to display objects during execution, or to output them to a file, or to perform garbage collection. A run-time system that has access to complete source type information from the compiler may not need to maintain such information independently, say in the form of object type-tags, in order to handle such applications. The compiler and the run-time system could be made to cooperate in automatically recreating and using this type information when needed. In Part II of this thesis, we will explore the technique of run-time *type reconstruction* that reconstructs the exact type of every object on demand without paying the overhead of type maintenance. Furthermore, we will explore ways in which static type information can be used to automatically generate specialized routines at compile-time for each data and control object within the program in order to perform such tasks.

1.2 Id: A Strongly-Typed, Layered Programming Language

The idea of using type information within the design of a high-level language, its compiler, or its run-time system is not new. But, very few systems make use of source type information right from the design of an application all the way down to its execution in a coherent manner. This research is geared towards such an integrated approach to managing type information in the context of the parallel programming language Id [Nik91], developed at the Computation Structures Group, Laboratory for Computer Science, MIT.

Id is a high-level, strongly-typed language and it uses the Hindley/Milner polymorphic type system and its automatic type inference mechanism [Mil78, DM82] at its functional core. Id also offers imperative data-structures (I-structures [ANP89] and M-structures [BNA91]) that cater to imperative styles of programming. Id is a layered language by design (see Figure 1.1). The language and its implementation can be divided into three distinct layers: the user-level functional layer, the system-level imperative layer, and the architecture-level implementation layer.

At the highest level of functionality, the Id language provides high-level constructs such as arrays, lists, tuples, higher-order functions, and user-defined algebraic types. Special syntactic constructs, such as array and list comprehensions and pattern matching are also provided. Applications manipulating these objects make use of system and user libraries that support or extend the functionality provided by the compiler.

The system-level layer consists of the Id kernel language. The primitive I-structure and M-structure datatypes provide the basic data-structuring and synchronized memory access mechanisms in this language. These primitive datatypes are used to represent all high-level data-structures. Loops and procedures constitute the basic control mechanism. The compiler translates high-level syntactic constructs such as pattern matching, and list and array comprehensions into primitive operations on kernel datatypes. The system and user libraries may also make use of these kernel constructs to implement high-level data-structures.

Finally, the architecture-level layer consists of the run-time system of the language and is responsible for implementing the Id execution model and managing the synchronized memory. The compiler also generates type information and run-time support code for garbage collection and source-level debugging that can be directly linked along with the object code to perform

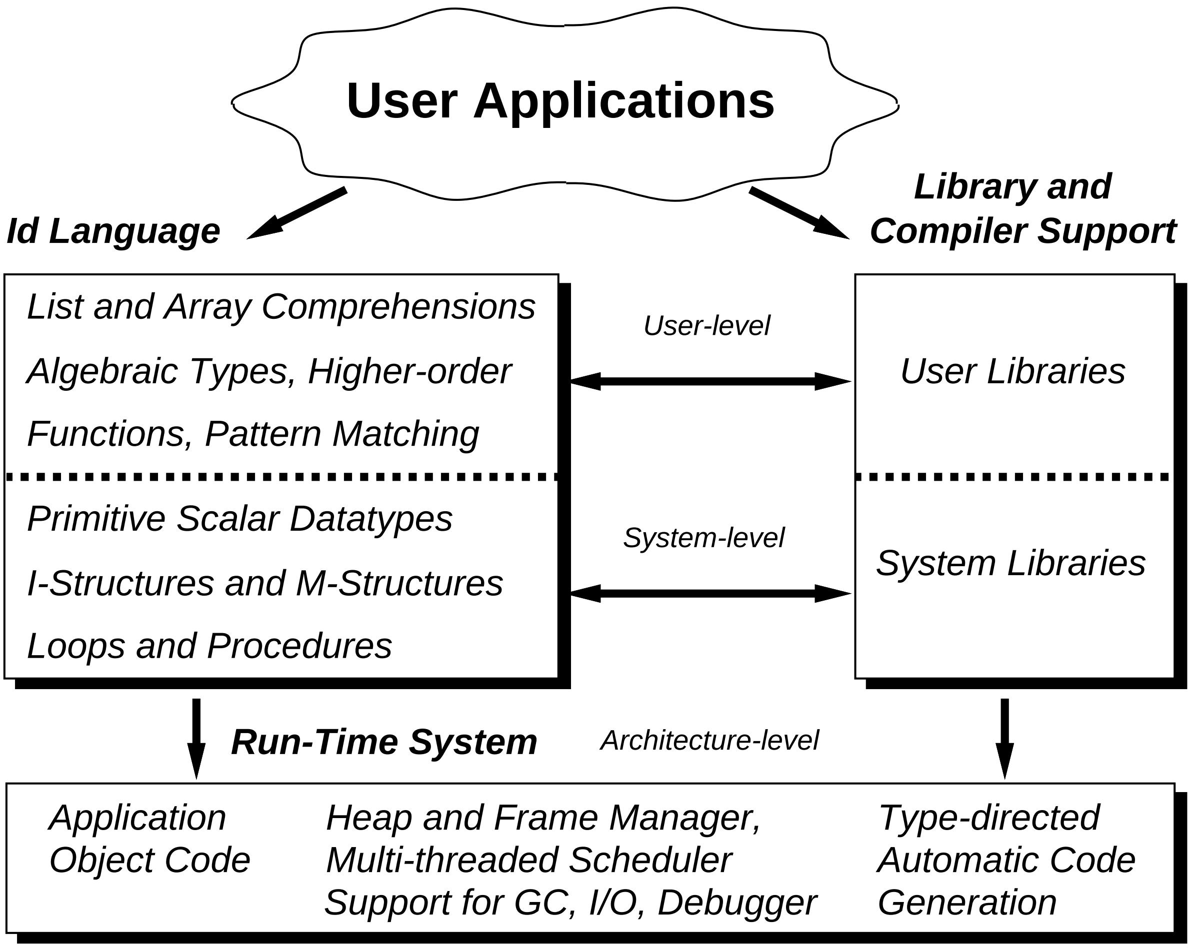

Figure 1.1: The Layered Design of the Id Language.

these auxiliary tasks during execution.

This layered design presents a very flexible interface to the application writer where more functionality can be added to the user-level simply by adding more system-level libraries written in the kernel language. The type system is responsible for clearly defining and enforcing the abstraction between the two layers so that polymorphic, functional behavior and simple reasoning can be preserved at the user-level. At the same time, the Id run-time system is able to map the system-level kernel language constructs onto the underlying target architecture in an efficient way, independent of the source language used.

1.3 Applications to Conventional, Imperative Languages

The functional encapsulation mechanism described in this thesis is not only applicable to higher-order, polymorphic languages like Id and Haskell, but also to conventional, monomorphic languages like C and Pascal. This mechanism allows safe conversion of mutable objects into read-only functional objects. This transformation is useful for both sequential and parallel versions of conventional imperative languages. We discuss some of these uses below.

The most important property of a functional object is that its value does not change during the course of execution of the program. Therefore, a functional object may be freely copied if necessary, or conversely, excessive copies may be freely eliminated. This property leads to obvious compile-time optimizations such as common sub-expression elimination, code-hoisting, and memory-fetch elimination that attempt to reduce the number of copies. This also permits unlimited caching of such functional data in a parallel machine without any risk of write-invalidation. In parallel systems using software-controlled shared-memory protocols [Nik94, FLR+94] this may directly translates into cheaper protocols for object access and migration.

While writing parallel programs, programmers often make implicit assumptions that a shared, mutable object may not be updated outside a given control block or that a particular processor may have exclusive access to a shared object without actually locking it. Such assumptions are usually based upon the implicit logic of the program and as such it may be quite difficult to prove their correctness. With a little help from the user in identifying such objects, the encapsulation mechanism described in this thesis can verify such assumptions automatically. This mechanism also allows making safe, unsynchronized access to such shared objects outside their encapsulated control-block because the objects are guaranteed to be read-only at that point.

Finally, conversion of mutable objects into functional objects also improves other compile-time analyses such as memory-aliasing analysis and loop-dependence analysis by clearly disambiguating between read-only and read-write data. This, in turn, may benefit automatic parallelization of sequential programs that make use of such analyses.

Thus, providing the ability to restrict the scope of side-effects to mutable data-structures translates into important optimizations at all levels of program design and implementation. This thesis provides the basic type-based framework for making such optimizations feasible.

1.4 Outline of the Thesis

1.4.1 Part I

This thesis is divided into two parts. In Part I (Chapters 2, 3 and 4), we describe a powerful type system that has the ability to encapsulate programs constructing mutable data-structures and view them as returning functional data-structures while guaranteeing that no more updates take place on the returned objects outside the encapsulation.

Chapter 2 is an informal and intuitive condensation of the major ideas in Part I. We introduce the problem by means of a simple example involving functional arrays in Id. We briefly survey the literature comparing various existing imperative type systems and informally describe our solution as an extension to one of the existing type systems. Then, we discuss "language-level issues" such as type-safety, polymorphism and non-mutability within our type system and how they interact with "system-level issues" such as space and time efficiency, parallelism, and memory synchronization. Finally, we describe specific strategies used in our type system that take care of these issues.

Chapter 3 describes the formal machinery and the soundness proof of our type system that is the main theoretical contribution of Part I. We start with the description of a small, imperative language containing simple mutable locations and a special language construct called `close` to convert them into immutable locations. We provide the dynamic and static semantics of this language in terms of relational axioms and inference rules and show their useful semantic properties. Then, we set up a semantic model that defines a consistent relation between values and their types. This relation maps read-write locations to mutable types and read-only locations to functional types. Finally, we prove a soundness theorem stating that the static and the dynamic semantics of our expression language are consistent with respect to each other. It follows immediately that the mutable objects that are successfully converted into functional objects under our type system, are never updated again dynamically.

Chapter 4 extends the formal machinery of Chapter 3 to complex datatypes such as arrays, tuples, functions, and general algebraic datatypes. We discuss how the user would syntactically specify the conversion of arbitrary, imperative data-structures into functional ones and how the compiler would automatically verify the soundness of this conversion. Finally we summarize the results of Part I and discuss directions for future research.

1.4.2 Part II

In Part II (Chapters 5, 6, 7 and 8), we study the technique of complete run-time type reconstruction and its various applications within the run-time system for Id.

Chapter 5 discusses some design issues that affect the use of type information within a run-time system. There are "language issues" such as strong *vs.* weak typing, and allowing polymorphism and higher-order functions in the language; "compiler issues" such as using static *vs.* dynamic typing, and type inference *vs.* type declaration; and "run-time system issues" such as using tagged *vs.* untagged object representation model, and using type maintenance *vs.* type reconstruction to obtain type information at run-time. We classify various programming languages on the basis of these issues. We also discuss our approach of complete run-time type reconstruction with an untagged object representation model and discuss some of its applications such as source debugging, tagless garbage collection and I/O.

Chapter 6 motivates the problem of compiler-directed polymorphic type reconstruction by means of examples and describes the technique informally. First, we describe the logical execution model of an Id program, dividing the work into compile-time, link-time, invocation-time and run-time. Then, we characterize the type information that needs to be recorded at compile-time to permit complete type reconstruction at run-time. Next, we informally describe how to analyze and translate the source program to propagate this information. Finally, we show the process of run-time type reconstruction using an example and discuss some optimizations.

Chapter 7 formalizes the concepts of Chapter 6 using a simplified kernel language for Id. This language is very close to the actual intermediate form used within the Id compiler. We present the analysis and program translation rules to generate and propagate all the necessary type information at compile-time. We also present a formal algorithm for type reconstruction and prove its correctness.

Finally, Chapter 8 discusses a full scale application of type reconstruction, tagless garbage collection. We describe a study that compares the performance of our type-reconstruction based garbage collection scheme with conservative garbage collection and a compiler-directed explicit deallocation scheme.

1.4.3 How to Read this Thesis

Both Part I and Part II are self-contained and may be read independently.

For Part I, Chapter 2 should be sufficient for readers that are only interested in understanding the problem, its context, and the intuitive ideas behind the proposed solution. Readers interested in the mechanics of the proposed type system and its extensions, possibly with a view towards implementing it, should look at the semantic machinery described in Sections 3.1

and 3.2 of Chapter 3, the extensions discussed in Chapter 4, as well as the type inference machinery described in Chapter 3 of [Ler92]. Of course, theoretical enthusiasts may want to go through all the detailed proofs provided in Chapter 3.

For Part II, Chapter 5 and Chapter 6 provide a general introduction to the idea of using type information at run-time, an intuitive description of the issues involved, and the technique of complete type reconstruction and its various applications. Chapter 7 is a must for readers interested in the detailed understanding and implementation of the type reconstruction mechanism, although the last section on the correctness proof of the reconstruction algorithm is mainly of theoretical interest. Finally, Chapter 8 provides a realistic perspective on the potential uses of this technique in the context of tagless garbage collection, and its cost trade-offs.

Part I

Types in Language Design: Functional Encapsulation

Chapter 2

Functional Encapsulation of Imperative Data-Structures

In this chapter, we study the problem of providing a suitable type abstraction mechanism between the user-level layer and the underlying system-level layer in a programming language. We introduce a new language construct called `close` that provides a statically verifiable, safe export mechanism for imperative data-structures from the system-level layer into the functional, user-level layer. We present several examples illustrating the usefulness of this construct and to discuss the technical issues involved in proving its soundness. We also compare our approach to other systems in the literature.

2.1 The Problem

Let us consider the problem of implementing *functional arrays* in Id that are homogeneous, non-mutable, polymorphic arrays. The library function `make_vector` creates a one dimensional functional array that memoizes a computation for a given index range as shown in the following example:[1]

Example 2.1:

```
def compute i = ... some large computation ...;
compute_memo = make_vector compute (1,10);
```

How is the function `make_vector` implemented? Operationally, one has to allocate an empty vector and fill it with the result of applying the given function to each index position. There are two possibilities. We could treat `make_vector` as a language primitive and hard-wire it within the compiler. Then, we would have to provide a slew of such primitive functions that define functional vectors, matrices, and higher dimension arrays, along with their common patterns of construction. Some languages (including Id) provide special array construction syntax called *array comprehensions* to alleviate this problem. While array comprehensions are convenient, they still do not cover many useful construction patterns. They also increase the complexity of the compiler and the language it must manipulate. Moreover, this solution does not apply to user-defined functional abstractions in addition to those already present in the language.

[1]All our examples use the Id language syntax [Nik91]. We will provide brief explanations as necessary. Function definitions in Id are introduced with the keyword `def`, all statements are terminated with a semi-colon (;) and application is by juxtaposition.

The other possibility is to provide an imperative kernel language using which `make_vector` and other array construction functions may be defined in a separate library. Special syntactic constructs like array comprehensions may also be desugared into this kernel language. The kernel language would support primitive operations such as simple arithmetic, allocating a vector, storing/fetching a value at a particular index of a vector, and simple control mechanisms such as iteration and procedure call. This is the approach taken in Id. This approach also enables a system programmer to implement arbitrary new abstractions without changing the language definition or the compiler. As an example, the `make_vector` function may be implemented in the array library as shown below:[2]

Example 2.2:

```
def make_vector f (l,u) =
   { a = i_vector (l,u);
     _ = { for i <- l to u do
               a[i] = f i };
    in a };
```

Here, `i_vector` is a kernel primitive that allocates an empty one dimensional I-structure array, which is filled with the result of applying the filling function to each index position. Under the non-strict, parallel evaluation model of Id, the array `a` is returned as soon as it is allocated; its filling loop executes in parallel. However, this does not create any race condition for the array because the I-structure protocol [ANP89] supports fine-grain producer-consumer synchronization on every memory location: multiple readers wait at an empty location until a single writer fills it with the desired value.

Nevertheless, as it stands, there are some technical problems with the above implementation. I-structures and M-structures are imperative constructs, *i.e.*, they can be assigned to,[3] whereas functional arrays are supposed to be non-imperative. Therefore, returning an assignable I-structure from `make_vector` is not appropriate. Furthermore, in the Hindley/Milner type system, imperative objects are allowed only a restricted form of polymorphism to ensure type-safety [Tof90]. Thus, the functional arrays implemented using I-structures in this manner would have restricted polymorphism, which reduces the utility of such library implementations.

Both problems described above may be solved by providing the ability to package the above implementation of the `make_vector` function into a type-safe, polymorphic, functional abstraction as required by its intended interface. In general, the kernel language should contain a general *type abstraction mechanism* that can properly encapsulate such imperative implementations of functional data-structures while ensuring their polymorphism and non-mutability outside the abstraction.

2.1.1 Abstraction and Polymorphism

It may appear that a conventional abstract datatype facility available in most modern languages should be sufficient for our purpose. Indeed, we could write a functional array datatype that is internally represented using I-structures and does not allow any mutation capability in its abstract interface. But, such an abstraction is still not completely satisfactory because it only

[2]All bindings within a `let`-block (enclosed within {}) execute in parallel. The bindings may be mutually recursive and their textual order is unimportant. An *underscore* (_) on the left-hand-side of a binding implies that the result of the right-hand-side expression is to be ignored. The result of the overall block is the value of the expression following `in`.

[3]Strictly speaking, I-structures are not mutable, since they have write-once semantics, but an empty I-structure can be filled with any value using assignment.

hides the internal data representation of the functional datatype, it does not automatically restore the full polymorphism of the functional datatype from the restricted polymorphism of its imperative implementation. This polymorphic strengthening of the datatype has to be done explicitly and under additional semantic analysis that guarantees its soundness. As we will see in Section 2.2, the treatment of type polymorphism is significantly more complicated by the presence of imperative constructs under the usual call-by-value semantics.

A radically different but equally interesting approach is to define a call-by-name semantics for the polymorphic objects within the kernel language and permit unrestricted polymorphism for imperative programs. In this case, the conventional data abstraction mechanism would be sufficient to hide the imperative implementation of a functional datatype. In this alternate semantics, called *polymorphism-by-name* [Ler93], the evaluation of a polymorphic object is suspended and each type instantiation re-evaluates the suspension in the current context to produce a fresh instance. In contrast, the usual ML-like polymorphism is called *polymorphism-by-value* where polymorphic objects are evaluated only once and the resulting value is shared among all its instances.

Leroy showed in [Ler93] that the naive Hindley/Milner typing rules are sound with respect to polymorphism-by-name semantics for imperative references and continuations. This approach is used in languages like Quest [Car89] and to a limited extent in CLU [LAB+81] where explicit type parameters are used to abstract and instantiate polymorphic objects. Unfortunately, suspension and re-evaluation of polymorphic objects destroys their sharing characteristics which are very important in the dynamic semantics of the Id language. Therefore, we would like to improve upon the abstraction characteristics of the polymorphism-by-value type systems that preserve such sharing.

In another case, Wright experimented with the type system of Standard ML by restricting polymorphism to only certain classes of syntactically recognizable values such as function declarations, constants, and known functional constructors [Wri93]. These functional values can be recognized statically and therefore can be generalized and shared safely. Mutable data-structures are always classified as dynamic entities and therefore can never be generalized. He showed empirical evidence that this restricted form of polymorphism is sufficient for a large class of existing Standard ML programs. Unfortunately, our ultimate goal is to provide functional and polymorphic view of dynamically created imperative data-structures for which this approach is entirely inadequate.

2.1.2 Outline of our Approach

We can divide our problem into two distinct phases. First, it is important to be able to give sound and accurate type semantics to imperative constructs in the kernel language. We must precisely capture the imperative types of mutable objects and propagate them with first class status while handling higher-order functions, storing into data-structures and passing them around as arguments and results.

The second phase involves presenting a functional view of the mutable objects to the end user. This may involve a semantic check on the part of the compiler (or the system programmer) as well as some sort of type conversion to convert the imperative types into functional types. The type system must ensure that fully functional and polymorphic behavior is projected through the abstraction both in static and dynamic semantics. We present a new language construct called `close` that achieves this functionality through the type system.

The interaction of polymorphism and imperative programming has been the subject of active research in the past decade [Dam85, Tof90, AM89, LW91, Ler92, TJ92, Wri92]. Several type systems have been proposed in the literature spanning a wide range of expressiveness and complexity. We present a brief survey in Section 2.2. Since our main task is to provide an encapsulation mechanism for imperative program fragments, we prefer to extend an existing imperative type system that meets our needs rather than design a new one. We have chosen the **Closure Typing** system proposed by Xavier Leroy in this thesis [Ler92] as a convenient starting point for our encapsulation extensions. We motivate this choice in Section 2.2.7. In Section 2.3, we informally describe the meaning and the use of the `close` construct *via* examples and discuss issues of type-safety, non-mutability, efficiency, parallelism, and memory synchronization in their context. Finally in Section 2.4, we present informal typing strategies that ensure the soundness of the `close` construct.

2.2 Imperative Type Systems

It is well known that the simple Hindley/Milner type system yields unsound typing when applied to mutable data-structures in the naive way. In this section, we briefly review this problem and describe some practical extensions to the type system that handle it to some extent.

2.2.1 Simple Hindley/Milner Type Inference

Consider the following example[4] in Id that emulates the `ref` construct of ML using a naive, functional Hindley/Milner type system:

Example 2.3:

```
type ref t0 = mkref !t0;                    % mkref :: ∀t0.t0 → (ref t0)

r = mkref identity;                         % r :: ∀t1.(ref (t1 → t1))

r!!mkref_1 = square;
_ = r!!mkref_1 true;                        % Dynamic Type Error!
```

The datatype `ref` defines a polymorphic constructor `mkref` that allocates a mutable cell and initializes it with a given value. The value contained in the cell can subsequently be updated by field assignment as shown above.

The type schemes[5] for the constructor `mkref` and the mutable cell `r` inferred by the naive type system are shown on the right. Note that the mutable cell `r` is given a polymorphic type which can be instantiated to $int \rightarrow int$ or $bool \rightarrow bool$ as desired. Thus, this example passes the type system even though it causes a run-time type-error, attempting to apply an integer function `square` to a boolean `true`. The problem is that the type of a mutable object should not be deemed polymorphic even if it initially contains a polymorphic value. This is because later such objects may be updated to contain values that do not possess the expected polymorphism. The type system must be aware of such mutable objects and keep track of their types in a sound manner.

One way to avoid such unsound polymorphism is to statically approximate the state of the mutable store and the set of objects stored within it. These (presumably) mutable objects are

---

[4]User-defined types and constructors in Id are introduced with the keyword `type`. A (`!`) in front of a constructor field denotes that it is mutable, and can be used in *M-take*/*M-put* (`!`) or *examine*/*replace* (`!!`) operations during field access. Fields are accessed by position using numeric suffixes (starting from 1) to the constructor name. Although Id has parallel semantics, our examples assume a sequential order of evaluation for simplicity.

[5]A *type-scheme* is a polymorphic Hindley/Milner type containing type variables, such as $t_0, t_1, \ldots$, some of which may be *bound* by the universal quantifier ($\forall$). Bound type variables may be substituted for different types in different contexts giving rise to polymorphic *type-instances*.

not allowed to have polymorphic types. The mutable store approximation needs to be updated whenever there is a possibility of allocating a new mutable object or updating an existing mutable object. This has to be achieved in a flexible but sound manner within and across function and local block boundaries.

Many type systems in the literature follow this general framework [Dam85, Tof90, LG88, JG91, Wri92, TJ92]. The various systems differ in their notion of a store abstraction and the amount of information propagated across function boundaries. An illustrative comparison of some of these systems is presented in [OJ91]. First, we will briefly describe two such systems that are simple extensions of the original Hindley/Milner type system and have been successfully used in practical programming languages. Then, we will describe two more recent type systems that are more complex but are much more powerful in dealing with higher-order functions.

2.2.2 Type System of Standard ML

In Standard ML [MT91, MTH90], type variables are syntactically classified into two separate categories: *imperative* type variables ($u_0, u_1, \ldots$) that may occur in the type of a mutable object at some stage of type inference and therefore implicitly model the abstract mutable store, and *applicative* type variables ($t_0, t_1, \ldots$) that can never occur in the type of a mutable object. Furthermore, since the evaluation of variables and λ-expressions (termed as *non-expansive* expressions) never generates any new mutable objects, imperative type variables occurring in their types are allowed to be generalized.[6] Whereas, applications and `let`-expressions (termed as *expansive* expressions) may allocate new mutable objects on evaluation, therefore imperative type variables occurring in their types are not allowed to be generalized. The resulting type system is sound [Tof90], easy to implement, and correctly rejects Example 2.3 as a type-error. To see this, note that under this scheme, storage allocating functions such as `mkref` always contain imperative type variables in their type-schemes because they allocate and return mutable memory locations. Thus, the type of `r` cannot be generalized since it will contain an imperative type variable that is created in an expansive expression (application).

One of the problems with this system is that the modeling of imperative objects is too simplistic. The imperative type variables model values contained in a mutable location rather than the locations themselves. This has the effect of "contaminating" the types of the values fetched out of mutable locations. Consider the following example:

Example 2.4:

```
def identity x = x;                          % identity :: ∀t0.t0 → t0
def identity′ x = (mkref x)!!mkref_1;        % identity′ :: ∀u0.u0 → u0

nil′ = identity′ nil;
x = 1:nil′;
y = true:nil′;                               % Static Type Error!
```

Although `identity′` is assigned a polymorphic type, it is still weaker than the type of the `identity` function. This is because the `identity′` function temporarily stores its argument within an imperative location. This contaminates the type of the returned result to be imperative and unnecessarily restricts its polymorphism as shown. We would have liked to assign the same type to both identity functions.

Another problem with this system is that the distinction of expansive and non-expansive

[6] *Type generalization* refers to the process by which some type variables occurring in a type are bound with a universal quantifier ($\forall$) converting that type into a polymorphic type scheme.

expressions is also very simplistic. In particular, this system cannot deal with higher-order or partially applied imperative functions. Consider the following example:

Example 2.5:

```
mkref′ = identity mkref;          % mkref′ :: u0 → (ref u0)

x = mkref′ 1;
y = mkref′ true;                  % Static Type Error!
```

The application of `mkref` to the `identity` function strips out its polymorphism because the type system deems this application as expansive whether or not any mutable reference was allocated within the `identity` function. This causes the unnecessary type-error to be flagged by the type system. The following example illustrates a similar problem for partial applications:

Example 2.6:

```
def imp_map f l =                 % imp_map :: ∀u0u1.(u0 → u1) → (list u0) → (list u1)
  { arg = mkref l;                % arg :: (ref(list u0))
    res = mkref nil;              % res :: (ref(list u1))
    in
     {while not (nil? arg!!mkref_1) do
       x:xs = arg!mkref_1;
       arg!mkref_1 = xs;
       res!mkref_1 = f x : res!mkref_1;
      finally reverse res!mkref_1}};

def fn_map f nil = nil            % fn_map :: ∀t0t1.(t0 → t1) → (list t0) → (list t1)
 |..fn_map f (x:xs) = f x : fn_map f xs;

list_identity = imp_map identity;          % list_identity :: (list u3) → (list u3)
u = list_identity (1:2:nil);
v = list_identity (true:false:nil);        % Static Type Error!
```

Just like `identity′` function in Example 2.4, the type-scheme assigned to the function `imp_map` in the above example contains imperative type variables because it uses mutable locations internally, while its functional version `fn_map` carries only applicative variables. Furthermore, when using `imp_map`, although no actual allocations take place until after its second argument, the type system has no way to determine this and it deems the first application to be expansive as well. This results in a non-polymorphic type for the `list_identity` function as shown. This problem of typing partial applications was fixed in part by the type system of Standard ML of New Jersey, which we discuss next.

2.2.3 Type System of Standard ML of New Jersey

The type system of Standard ML of New Jersey [AM89] assigns an integer rank to each imperative type variable. We write these ranks as superscripts on the type variables. A rank 0 imperative type variable u^0 occurring within the type of an expression at the top-level indicates that the type of some existing mutable object already contains u^0 and therefore u^0 should not be generalized. Such a type variable is said to have entered the mutable store typing. A positive rank-d ($d > 0$) imperative type variable u^d occurring within a function type denotes the number of application after which u^d will enter the store typing. Therefore u^d is allowed to be generalized for up to $d - 1$ partial applications involving the function type where each partial application reduces its rank by one. This scheme is extended to typing objects enclosed within

λ-abstractions by keeping track of the number of application necessary to make them enter the store typing. The resulting type system is slightly more complex than Standard ML but still relatively easy to implement and has been recently shown to be sound [HMV93].

Without going into details, it should be clear that this modification handles the function `list_identity` in Example 2.6 quite well. The type of the function `imp_map` is now inferred to be $\forall u_0^2 u_1^2.(u_0^2 \rightarrow u_1^2) \rightarrow (list\ u_0^2) \rightarrow (list\ u_1^2)$, where the superscript 2 denotes that the actual allocation of imperative objects in the function's body does not take place until after the second application. Therefore, the type of `list_identity` is inferred to be $\forall u_3^1.(list\ u_3^1) \rightarrow (list\ u_3^1)$, where the superscript 1 denotes the fact that one more application of this function will create some fresh mutable memory locations.

Unfortunately, the simple ranking mechanism outlined above is still not sufficient to deal with imperative higher-order applications as shown in Example 2.5. The type system does not have any way to characterize when and how to incorporate "potentially" imperative type information from arguments of higher-order functions within their final result. Therefore, the type system must conservatively assume that all imperative functions generate mutable objects when passed as arguments to higher-order functions. The following comparison illustrates this point:

Example 2.7:

```
def ap_nil f = f nil;                     % ap_nil :: ∀t0t1.((list t0) → t1) → t1

foo = ap_nil mkref;                       % foo :: (ref (list u1^0))
mkref' = identity mkref;                  % mkref' :: u2^0 → (ref u2^0)
```

Here, the imperative function `mkref` is passed as an argument to two polymorphic functions `ap_nil` and `identity`. In the first case (identifier `foo`), the type of the application is correctly inferred to be non-polymorphic because it actually creates a fresh mutable reference. But in the second case (identifier `mkref'`), the type of the application is unnecessarily non-polymorphic because the `mkref` function is never applied within the body of the `identity` function. The type of the identifier `mkref'` should in this case be $\forall u_2^1.u_2^1 \rightarrow (ref\ u_2^1)$, which is identical to the type of the constructor `mkref` in this system. The problem is that the type system has no way of knowing that the function `ap_nil` applies its parameter `f` to one argument and therefore may potentially create mutable references, while `identity` passes its parameter unchanged and therefore cannot create any mutable references. Hence, the type system must conservatively assume that all imperative functions create mutable references when passed as arguments.

The formalization for the above type system presented in [HMV93] is somewhat more powerful than the SML/NJ compiler implementation and it can deal with the above situation correctly. Although, it requires a more complicated mechanism for rank book-keeping and uses rank variables instead of fixed integral ranks. It also entails a more complicated type unification mechanism that needs to resolve algebraic constraints on rank variables. The interested reader is referred to [HMV93].

2.2.4 Limitations of the Standard ML Type Systems

Although the two type systems presented above cover a lot of practically useful cases of imperative programming, they are still not sufficiently powerful for our purposes. Ultimately, we intend to smoothly convert mutable types into functional types, so our type system must not only propagate the mutable type information properly where necessary, but also keep it self-contained and easy to manipulate. The problem of type variable contamination as shown in Examples 2.4 and 2.6 is a serious one in this regard. None of the systems presented above have the ability to assign the same polymorphic type to functions `identity` and `identity'` or functions `fn_map` and `imp_map`. At some observational level, these functions are equivalent but the internal implementation of the imperative versions shows up in their type and hence they are not interchangeable with respect to these type systems.

Another fundamental problem with the above systems is that they concentrate on modeling "imperativeness" of objects only to the extent it affects their type polymorphism. For instance, there is no difference between the type of a record that contains a functional integer field and the one that contains a mutable integer field. Since we are ultimately interested in approximating dynamic mutability of all objects by means of their static types (whether polymorphic or not), the partial modeling offered by the type systems above is also unsatisfactory for our purposes.

Both observations above show that tying the "imperativeness" of mutable objects to the kind of type variables contained in their types is rather simplistic and imprecise. We will now look at some type systems where this information is tracked independently, leading to a much more complete and cleaner characterization of imperative objects.

2.2.5 Effect Systems

Effect systems are a broad class of polymorphic typing systems that use static type-checking and inference techniques to model the dynamic behavior of programs written in imperative languages [Luc87, LG88, TJ92, Wri92]. Originally, such systems were used to collect and propagate side-effect information across program fragments for compiler optimizations and parallelization [LG88]. One such type and effect system was successfully used in the FX-87 language [GJLS87] which supported explicitly declared type polymorphism.

More recently, automatic type and effect inference techniques have been developed [TJ92, Wri92] that use the effect propagation mechanism to infer types that model polymorphic imperative objects more accurately than the systems given above. As we will see shortly, such type and effect systems can be viewed as a logical extension of the type systems described above.

Effect Analysis

Probably the most appealing aspect of effect systems is their uniform and integrated mechanism of type and effect information propagation across all function and local block boundaries. The key idea is that every expression generates a read/write/allocate *effect* which is accumulated along with its type. The effect of the body of a function, parameterized by the effect of its formal arguments, is summarized as the *latent effect* of the function on the arrow type-constructor ($\rightarrow$) in its type. Functions by themselves have no immediate effect. Unknown latent effects for functional parameters of higher-order functions are modeled using *effect-variables*. This effect parameterization permits a clean way of computing the overall effect of a function application by instantiating its latent effect by the effects of its actual arguments. The effect information propagated and accumulated in this manner may then be used to accurately identify the creation of polymorphic imperative objects and avoid their unsafe generalization.

In one of the simpler effect systems proposed by Wright [Wri92], all type variables present in the type of a freshly allocated mutable data-structure are collected as part of the effect of that allocation. The explicit effect computation and propagation mechanism obviates the need to mark such type variables as imperative. Unsound typings are then avoided by disallowing generalization of type variables that occur in the immediate effect of an expression. This system still does not deal with the issues of imperativeness and type polymorphism independently, but at least the information flow across higher-order function boundaries is improved because of the effect propagation techniques.

As an example, consider the function `fn_map` shown below:

Example 2.8:

`fn_map` $:: \forall t_0 t_1 f_0.(t_0 \xrightarrow{f_0} t_1) \xrightarrow{\phi} (\mathit{list}\ t_0) \xrightarrow{f_0} (\mathit{list}\ t_1)$

```
def fn_map f nil = nil
 |..fn_map f (x:xs) = f x : fn_map f xs;
```

`mkref` $:: \forall t_0.t_0 \xrightarrow{\{alloc(t_0)\}} (\mathit{ref}\ t_0)$

`ref_list` $:: (\mathit{list}\ (\mathit{ref}\ (\mathit{list}\ t_2)))$ with immediate effect $\{alloc(\mathit{list}\ t_2)\}$

```
ref_list = fn_map mkref (nil:nil);
```

$f_0, f_1, \ldots$ are effect-variables which may be substituted for any effect. ϕ denotes the null effect. Effect-variables are allowed to be generalized and instantiated just like type variables. The type of the function `fn_map` illustrates the use of these effect variables. The latent effect of the mapped function is captured in the effect variable f_0 that is exposed in the final effect of the `fn_map` function.

The example also shows the type of the reference allocator function `mkref`. The latent effect appearing over the arrow ($\rightarrow$) shows that the function allocates a mutable object of type t_0. As shown in the example, the effect of mapping `mkref` to a polymorphic list instantiates and exposes its latent effect of allocating mutable cells containing polymorphic objects. Since t_2 is present in the immediate effect of the expression creating `ref_list`, it cannot be generalized.

This system infers the type-scheme $\forall t_0 t_1 f_0.(t_0 \xrightarrow{f_0} t_1) \xrightarrow{\phi} (\mathit{list}\ t_0) \xrightarrow{\{t_0,t_1\}\cup f_0} (\mathit{list}\ t_1)$ for the function `imp_map` of Example 2.6 (*c.f.* `fn_map` of Example 2.8). Note that the first application has no effect, and the second application records the effect of allocating new local memory references for internal identifiers `arg` and `res` (as a set of type variables t_0, t_1 occurring in those reference types) as well as the effect of applying the argument function `f` (captured *via* the effect-variable f_0). Thus, in this system, partial curried applications do not expose the final effects prematurely, but the problem of type contamination by unnecessarily exposing local effects still exists.

Principal Types and Minimal Effects

In order to compute the type and the effect of every expression automatically and efficiently, one must show that the system admits unique principal types and effects for expressions and that they are computable using an efficient inference algorithm. At least two effect-based systems [TJ92, Wri92] propose such inference mechanisms based on structural unification [Rob65]. The effect system of FX-91 [GJSO91] uses the more complex algebraic unification [JG91] which permits unification *modulo* algebraic identities such as associativity and commutativity. This provides more expressive power to the inference system, albeit at the cost of simplicity and efficiency. Here, we will only discuss the inference system based on standard structural unification.

The basic idea is to compute the principal types of expressions in the usual way using the standard Hindley/Milner type inference mechanism while accumulating a set of constraints for the latent effect of all the function types in the program. Then, this constraint set is solved separately to obtain the minimal effect of each function in the program. This process is not completely straightforward because of the possibility of cyclic constraints created due to mutually recursive functions. The following examples illustrate this problem[7]:

[7]The latent effects of functions are represented in this system by a constrained effect-variable. The constraints

Example 2.9:

```
def f0 x = f1 x;                  % f0 :: t0 -f0-> t1 with {f0 ⊒ f1}

def f1 x = f0 x;                  % f1 :: t0 -f1-> t1 with {f1 ⊒ f0}

def g x = { a = mkref x;          % g :: t0 -f0-> (list (ref t0)) with {f0 ⊒ ({t0} ∪ f0)}
            in a:(g x) };

def h x = { a = mkref h;          % h :: int -f0-> int with {f0 ⊒ {int -f0-> int}}
            in x+1 };
```

Type annotations:

- `f0` :: $t_0 \xrightarrow{f_0} t_1$ with $\{f_0 \sqsupseteq f_1\}$
- `f1` :: $t_0 \xrightarrow{f_1} t_1$ with $\{f_1 \sqsupseteq f_0\}$
- `g` :: $t_0 \xrightarrow{f_0} (\mathit{list}\ (\mathit{ref}\ t_0))$ with $\{f_0 \sqsupseteq (\{t_0\} \cup f_0)\}$
- `h` :: $\mathit{int} \xrightarrow{f_0} \mathit{int}$ with $\{f_0 \sqsupseteq \{\mathit{int} \xrightarrow{f_0} \mathit{int}\}\}$

Minimal effects in the above cases are computed by combining the effects of all cyclic constraints into one and finding the least assignment to effect-variables (starting from the null effect ϕ) that would satisfy all the constraint inequations. Thus, functions `f0` and `f1` in the above example are each assigned the null effect ϕ and the function `g` gets the effect $\{t_0\}$. The function `h` represents an interesting case. Depending on the desired semantic interpretation of effects, the least effect satisfying this constraint may be taken to be infinite and such expressions may be classified as ill-formed (system [TJ92]), or this constraint may be simplified to $\{f_0 \sqsupseteq \{f_0\}\}$ which yields the null effect ϕ as the minimal solution (system [Wri92]).

Region Analysis and Effect Masking

Some effect systems also carry out a *region analysis* of memory allocation and sharing [LG88, TJ92]. The static description of an expression also summarizes a conservative approximation of the memory regions (locations) manipulated within the expression, in addition to its type and effects. If a set of regions is found to be purely local to an expression, *i.e.*, if these regions are not accessible through a free variable of the expression, and if they are not exported *via* the result of the expression, then the effects associated with those regions may be erased from the overall effect of that expression. The idea is that only certain "observable" effects on "visible" regions need to be kept, the rest may be safely erased without affecting the semantics of the program. This is known as *effect masking*. This analysis may be able to mask all the side-effects to internal data-structures of a procedure which largely alleviates the problem of type contamination. In this sense, this scheme is capable of automatically assigning purely functional types for some classes of imperative programs. For example, these stronger systems are able to infer the same type for `imp_map` and `fn_map` (namely, $\forall t_0 t_1 f_0 .(t_0 \xrightarrow{f_0} t_1) \xrightarrow{\phi} (\mathit{list}\ t_0) \xrightarrow{f_0} (\mathit{list}\ t_1)$) since the mutable references created within `imp_map` can be masked.

Region analysis requires a lot of book-keeping to maintain a very fine static notion of the mutable store. The benefit of obtaining this additional region information and performing effect masking has to be weighed against the extra complexity required to do these analyses in a practical language implementation[8]. Furthermore, effect masking does not cover all cases of effect erasure that we are interested in. For example, the effect generated by the `make_vector` function of Example 2.2 can not be masked since the mutable vector is being returned as the result of the function. The user can still update this vector and destroy the type polymorphism that might result by erroneously masking this effect.

---

are of the form (*effect-variable* $\sqsupseteq$ *effect*) which means that the effect on the right hand side is a lower bound on the actual effect denoted by the effect-variable on the left hand side.

[8]Indeed, region analysis was dropped from FX-91 language [GJSO91] which is a more recent version of FX-87 [GJLS87].

Analysis of Effect Systems

On the whole, effect systems seem to be a powerful tool to summarize a variety of dynamic behaviors of programs accurately. But we still have to extend the effect masking analysis to meet our original goal, which is to transparently encapsulate imperative programs into functional abstractions. We also anticipate that external factors such as a user-declared functional interface will play an important role in guaranteeing type-soundness in our system in spite of otherwise non-maskable imperative effects in the program. None of the existing effect systems incorporate such information. In Section 2.4, we will explore some of these ideas where a powerful type system is combined with user-supplied information while trading off some of its power for speed and simplicity.

2.2.6 Syntactic Closure Typing System

All the type systems we have seen so far, model the state of the dynamic mutable store and the operations performed on it using some static approximation, and then use that information to identify the objects that can safely be assigned polymorphic types. Instead of approximating the dynamic behavior of the program, Leroy and Weis [LW91] introduced a more direct, syntactic way of identifying and safely typing mutable objects using an extension of the Hindley/Milner type system. We discuss their technique below.

Syntactic Analysis

The key idea in the scheme proposed by Leroy and Weis is that the static type of a complex object can be used as a clue to its structural shape and dynamic properties (such as mutability) of its various components. For example, an *i_vector* type represents an assignable, I-structure array, while a *vector* type represents a functional array. This is exactly the information required to decide what parts of that object's type can be safely generalized. Note that this information is independent of when/where/how the object was created in the program and depends only on its static type structure. This analysis relies on the assumption that the type of an object remains visible from all places within the program where that object may actually end up. Then the generalization scheme is simply that the type variables present within an assignable portion of a type (such as t_0 within the type $(i\_vector\ t_0),(list\ t_1))$ are considered to be *dangerous* and are not generalized, while all other type variables occurring elsewhere in the type (such as t_1) are allowed to be generalized.

The key assumption in the above scheme is that all objects can be viewed as data-structures whose component types are reflected back in the type of the overall object. In particular, objects captured inside the environment part of a function closure must also be made visible in the type of that function. Otherwise the mutability information of a datatype could be lost by placing it within a function closure. The following example illustrates this point (*c.f.* Example 2.3):

Example 2.10:

```
def fnref x =                          % fnref :: ∀t0.t0 → (void -(ref t0)-> t0, t0 -(ref t0)-> void)
  { r = mkref x;
    def read () = r!!mkref_1;
    def write y = { r!!mkref_1 = y };
   in
    read,write };

reader,writer = fnref identity;
```

```
_ = writer square;
_ = reader() true;                    % Static Type Error!
```

The function fnref emulates the functionality of the mutable constructor mkref of Example 2.3 by creating separate read and write handles to a shared mutable reference r. In the scheme proposed by Leroy and Weis, the function type-constructors ($\rightarrow$) of read and write functions are augmented with *closure types* that expose the types of objects captured within their closure environments. Without closure types, it is impossible to tell from the types of the read and write functions that they share a mutable reference. Thus, closure types help in identifying hidden dangerous type variables and therefore avoid their unsound type generalizations. In the above example, when the function fnref is used to create the reader and writer handles to a hidden mutable reference to identity, their non-empty closure types correctly restrict their types to be non-polymorphic and the type-error can be detected.

In general, closure types for a function must keep track of the type of all the free variables occurring within the function body, whether such types are dangerous or not. This is because such free variables may correspond to formal parameters of an enclosing function that may ultimately be instantiated with a mutable object at some application site. The following example illustrates this point:

Example 2.11:

```
def K x = {fun y = x};                % K :: ∀t0t1. t0 → (t1 -t0-> t0)
f = K (mkref identity);               % f :: ∀t1. t1 -ref(t2→t2)-> ref (t2 → t2)
```

$K :: \forall t_0 t_1.\, t_0 \rightarrow (t_1 \xrightarrow{t_0} t_0)$

$f :: \forall t_1.\, t_1 \xrightarrow{ref\,(t_2 \rightarrow t_2)} ref\ (t_2 \rightarrow t_2)$

The type of function f correctly generalizes t_1 and not t_2 because t_2 occurs under a mutable type in its closure. This was possible only because we correctly kept track of the type t_0 of the free variable x in the closure type of the body of the function K.

Type Soundness and Type Inference Mechanism

Leroy developed this idea in his thesis [Ler92] showing the soundness of a type system with closure types with respect to the dynamic operational semantics of a ML-like language with higher-order functions. He also showed a type inference algorithm based on this type system that is sound and infers principal types and closure types.

The type inference for closure types turns out to be very similar to effect inference. A new class of variables called *closure extension variables* model the unknown closure types of higher-order functions just like effect-variables. There is some flexibility in deciding what closure type information really needs to be kept and what can be discarded. For example, it is possible to keep only *dangerous* and certain *visible* type variables within a closure type of a function instead of recording the full types of all its free variables. The algebra of Hindley/Milner types also has to be extended to incorporate extensible sets of closure types, including ways to compute dangerous type variables of a type and performing type substitution within closure types. The type inference mechanism then computes the usual Hindley/Milner types for all objects while accumulating a set of closure types for every function using simple structural unification. The interested readers may refer to [Ler92] for details.

Analysis of the Closure Typing System

Leroy's syntactic system also succeeds in giving the same polymorphic type to imp_map and fn_map functions just like the effect-based systems with effect masking. In his thesis [Ler92], Leroy makes some interesting comparisons of the expressive power of the various systems we have seen so far. His system turns out to be incomparable to the effect-based systems in power. This is not too surprising because his approach is semantically very different from the effect-based systems.

2.2.7 Choosing an Imperative Type System

As mentioned earlier, we have chosen the closure typing system of Leroy as a starting point for the typing extensions proposed in this thesis. In this section, we attempt to motivate this choice in the context of the various type systems we have seen above.

The real choice is between an effect-based system (Section 2.2.5) or the closure typing system (Section 2.2.6). Type systems of Sections 2.2.2 and 2.2.3 and their extensions are either too simplistic in that they do not deal with higher-order functions properly or they suffer from the problem of type contamination.

A requirement imposed by our ultimate goal to selectively convert some imperative objects into functional ones is that we should be able to uniquely label groups of imperative objects, recognize them independent of other objects, and track their movement within the program. Some sort of region-based analysis is necessary for this purpose. Either an effect-based system with regions may be used, or we may have to extend the closure typing system with regions.

The contrast between the closure typing and the effect-based approaches may be understood by examining the way in which imperative type information is collected and propagated. In the closure typing system, the type of an object directly describes its imperative or functional composition. This is purely static, locally determinable, object-based information. This property is extended even to functions where closure types are added to function types in order to describe the data captured within the closure environment. This is very appealing because at any given moment, all the relevant information about an object is available from its type wherever that object (and hence its type) travels. We say that this approach is *data-driven* since it keeps the relevant properties directly attached to the types of the data objects.

On the other hand, in an effect-based system the object themselves are not classified as imperative or functional. We collect the operations performed on various kinds of objects in a separate effect. Such effects are carried over object manipulators (functions) as their latent effect. At any given moment, the properties of an object can be ascertained indirectly by examining the kind of effects it is participating in. Such a system is very good in summarizing dynamic properties of program fragments rather than describing the data itself. We say that this approach is *control-driven* since it keeps the relevant properties attached to types of control objects (functions).

For our purpose, the *data-driven* approach is more direct and natural, since we are interested in determining and manipulating the imperative or functional nature of data objects directly. We need not separately keep track of the dynamic properties of the functions manipulating these objects. A sound, functional abstraction of an object can be built simply by changing the type of the object regardless of the way it is computed. Additional user information about an object should also be easy to incorporate into this system as long as we can show that such information preserves the type-safety of the static semantics. Due to these reasons we have chosen this type system as the basis of our extensions for converting imperative objects into functional ones, which we refer to as "closing" the imperative objects.

2.3 Closing Imperative Data-Structures

In this section we will informally describe what we mean by "closing" an imperative object and discuss several technical issues arising out of it.

2.3.1 A Proposal for "Close"

We observed in Section 2.1 that the returned array from Example 2.2 is mutable and must be assigned a restricted form of polymorphism. This restriction is necessary to achieve the desired type-safety in the following example:

Example 2.12:

```
def fill n = identity;

a = make_vector fill (l,u);        % a :: (i_vector (t0 -> t0))
a[i] = square;                     % a :: (i_vector (int -> int))
_ = a[j] true;                     % Static Type Error!
```

The Hindley/Milner type of the returned array is shown on the right where the type variable t_0 occurs free and is not generalized. The assignment "`a[i] = square;`" refines the type of the array `a` as shown which correctly generates a type-error on encountering the subsequent application to `true`. This is necessary because the indices `i` and `j` may be turn out to be the same at run-time, in which case this application would lead to a run-time type-error. All imperative type systems in the literature [Dam85, Tof90, AM89, LW91, Ler92, TJ92, Wri92] catch this type-error at compile-time by restricting the polymorphism of imperative objects in one way or another.

"Close" as a Type Converter

Although the above behavior for `make_vector` is correct, ultimately, we want it to behave like a functional array constructor that returns a non-mutable, polymorphic array. The interesting observation is that if we *convert* the type of the returned array from `make_vector` to be the functional type constructor *vector*, then all mutation operations on it are automatically made illegal since it must now be viewed as a functional object. In this case, we would have flagged a semantic error at the assignment "`a[i] = square;`". Since no more mutations are allowed on the array `a`, we may be able to safely generalize its type with respect to t_0. Henceforth, we will call this type conversion and subsequent type generalization operation as "closing" an imperative object.

We can rewrite the `make_vector` implementation to reflect the above strategy:

Example 2.13:

`make_vector` $:: \forall t_0.(int \rightarrow t_0) \rightarrow (int, int) \rightarrow (vector\ t_0)$

```
def make_vector f (l,u) =
  close { a = i_vector (l,u);
          _ = { for i <- l to u do
                    a[i] = f i };
         in a };
```

The `close` construct in this implementation is intended to be a special form that captures our notion of closing an imperative object. It provides an alternate "functional view" for the imperative object. Users may use this construct in their programs to convert an imperative data-structure (like the array `a` above) into a functional one. Or, such conversions may be issued automatically by a compiler while desugaring high-level functional constructs into low-level imperative program fragments. In either case, the type of the object being closed is converted from a mutable to a non-mutable type constructor that permits its subsequent type generalization.

"Close" as an Encapsulator

An important point to observe in Example 2.13 is that the `close` construct *encapsulates* the entire computation that allocates, fills, and returns the array `a` rather than acting merely as a *marker* for the array to be closed. Treating the `close` construct as an encapsulator clearly identifies the "scope" of the imperative operations being performed on the array `a`. Within this scope, imperative operations on the array are permitted, while outside this scope, the array is viewed functionally. This notation is useful both to the user, by providing a clear visual separation between the imperative and functional parts of the program, and to the compiler, that may need to compile these parts differently as well as verify the correctness of the `close` operation automatically. This implies that the following two expressions are not equivalent:

$$\texttt{close}\ exp \not\equiv \{\ \texttt{x = } exp\texttt{; in close x}\ \}$$

Here, *exp* stands for an imperative program fragment that allocates and prepares an imperative object for closing. The `close` construct on the left-hand-side behaves like an encapsulator: it encapsulates the entire program fragment that builds the object imperatively and then returns it with a functional view. There is a clear separation between the imperative and the functional views of the object. While, the `close` construct on the right-hand-side identifies the object to be closed but it does not clearly identify the program region where the `close` operation should take effect. Thus, it becomes difficult for the type system to verify the correctness of the `close` operation. The importance of this distinction will become clear shortly.

As a matter of notation, when only some of the objects being returned from an expression are to be closed, we specify it in a type annotation for the entire expression, where some of the components are only partially supplied:

Example 2.14:
```
close { a = i_vector (1,n);
        b = i_vector (1,n);
        ...
       in a, b } :: (vector _),_;
```

The underscore (_) within the annotation implies that the `close` operation does not apply to that particular component of the result. All other components of the result are closed according to the type specified. Thus, in the above example, the array `a` is closed into a functional vector while the array `b` remains open. The contents of the array `a` also remain unaffected. The exact details of this specification appear in Chapter 4.

2.3.2 Guaranteeing Type-Safety

The problem of closing imperative objects is not simply a matter of type conversion as it might appear from the above discussion. Note that the closing operation is type-safe only if the object does not *escape* the scope of the imperative implementation in any other way except *via* some controlled, safe paths. We saw above that if the only access to a polymorphic imperative object is through the returned result, then a type conversion allows us to do a type-safe generalization later on. But there are several other ways in which an object might escape a given scope, some of which are shown in the following example. Note that using the `close` construct as an encapsulator helps in identifying the escaping objects clearly:

Example 2.15:

```
def escape_1 n =
  close { a = i_vector (1,n);
          ...
          b[1] = a;                  % Storing into an external data structure
         in a };

def escape_2 n =
  close { a = i_vector (1,n);
          ...
         in a, a } :: (vector _),_;  % Returning unconverted object directly

def escape_3 n =
  close { a = i_vector (1,n);
          ...
          def g i v =                % Returning a write handle within a closure
            { a[i] = v; in v };
         in a, g } :: (vector _),_;
```

In function `escape_1`, a reference to the locally allocated imperative array `a` is stored into an external array `b`. The type of the array `b` is constrained to be (*i_vector* (*i_vector* t_0)) implying that the array `a` is still accessible in its open form through this indirection. In function `escape_2`, two references to the same array are returned: one is closed and the other is left open according to the specified annotation pattern. Mutations *via* the open reference will affect the type-safety of the closed version. The same effect is achieved in the function `escape_3`, although it is disguised in the form of a function that provides a write handle to the array being closed.

The essential problem in the above examples is that it is safe to close a polymorphic mutable data-structure only if it is guaranteed that no write handle pointing to that object remains accessible to the user after it has been closed. Otherwise, the subsequent functional behavior implied by the `close` operation and its possible type generalization will both be unsound.

All the imperative type systems in the literature automatically take care of such cases by avoiding generalization of imperative objects at all times. The trouble arises when we wish to force the type system to accept a functional, polymorphic type for an imperative object as implied by the `close` construct in the above examples. Then, either the user must be held responsible for the type-safety of the resulting program, or it becomes the responsibility of the type system (the compiler) to automatically verify the soundness of this transformation and reject the unsafe cases.

2.3.3 Guaranteeing Non-Mutability

Note that type-safety is an issue only for *polymorphic* imperative objects, *i.e.*, imperative objects that have some potentially generalizable type-variables in their type. This is because the usual typing rules would ensure that all values assigned to monomorphic mutable objects would have compatible types. For example, all the functions in Example 2.15 would be type-safe if assumed monomorphic even if the array being returned was subsequently mutated.

However, our intended meaning of the `close` operation is more than simply ensuring safe type generalizations. We want to enforce *non-mutability* of the returned data-structure which is a much stronger property of dynamic semantics compared to the weaker property of merely avoiding run-time type-errors (type-safety). For polymorphic objects, non-mutability implies type-safety and vice versa, but that is not the case for monomorphic objects. As the preceding discussion shows, ensuring non-mutability involves a simple form of *escape analysis* on the part of the compiler which is conventionally performed using dataflow analysis or abstract interpretation [GP90, GPG91, HI89]. Indeed, all the imperative type systems in the literature concentrate on the issue of type-safety alone.

In our case, we intend to model such simple form of escape analysis for free using the existing machinery of our type system that is already required to ensure its type-safety. Our machinery ensures true functional semantics for successfully closed objects, *i.e.*, such objects are guaranteed to be side-effect free and can participate in compiler optimizations such as common sub-expression elimination and code-hoisting that depend upon the objects being functional. Thus, the `close` construct serves as a true interface between the low-level, imperative layer and the high-level functional layer of the language.

2.3.4 Efficiency and Parallelism

Consider the following example adapted from [BNA91] that builds a n-bucket functional histogram of objects stored in a binary search tree. The search tree datatype is also shown below for convenience:

Example 2.16:

```
type tree t = leaf | node t (tree t) (tree t);

def histogram t n =
    close { a = m_vector (1,n);
            _ = { for i <- 1 to n do
                      a![i] = 0 };
            _ = accum t a n;
            ---
           in a };

def accum leaf a n = ()
 |  accum (node x l r) a n =
    { i = hash x n;
      a![i] = a![i] + 1;
      _ = accum l a n;
      _ = accum r a n; };
```

The `histogram` function allocates an empty mutable vector with n buckets and initializes each of the buckets to zero. The `accum` function uses pattern-matching to traverse the tree structure recursively and increments the count in the appropriate bucket.[9]

A couple of important observations can be made about the above example. First, all accumulations are made to the same mutable array which is closed and returned at the end. No copying is involved during accumulations or at the time of returning the final array. Most

[9] The notation "`a![i]`" in Id denotes *M-take/M-put* operations on mutable arrays with read-and-lock/write-and-unlock semantics. The notation "`---`" denotes a local *barrier*. All the computation above the barrier must terminate before any of the computation below the barrier is allowed to proceed. See [BNA91, Bar92] for details.

strongly-typed systems would only allow creating an internal mutable array to which accumulations are made, then copy the final tallies to a functional array which is returned. Hence, overall functional behavior is achieved at the cost of copying the final data-structure which may be quite expensive. The `close` construct automatically achieves the functionality without sacrificing the efficiency in such cases.

Second, all computations in Id are performed in parallel by default, constrained only by data-dependencies. In the above example, the histogram initialization and the entire tree accumulation can potentially be done in parallel. The `close` construct places no restrictions on the kind of parallel activities that can occur within the encapsulated expression — it simply closes and returns the final result. In a purely functional setting, some compilers would perform extensive destructive update analysis, linearity analysis, use linear type systems, abstract datatypes or monadic language constructs [Blo89, Wad90, Hud92, PJW93, LPJ94] to determine that the histogram may be safely single-threaded through the computation and hence modified in place. Not only does this require a lot of compiler analysis, but single-threading the computation completely destroys the parallelism inherent in the problem.

2.3.5 Termination of Side-Effects before "Close"

Example 2.16 illustrates another important point. Given the parallel execution model of Id, we must wait until all the accumulations have completed before closing and returning the histogram array in order to guarantee that the returned array is not updated anymore. This is ensured by inserting a *local barrier* (`---`) before returning the histogram which waits for all the computations before the barrier (issued in the current scope) to terminate before proceeding to the computations after the barrier. The barrier may be considered as an independent synchronization operation necessary for closing mutable objects in the presence of parallel updates (as shown here), or it could be taken as part of the `close` operation itself. In the latter case, the `close` construct would behave like a strict encapsulator that releases the closed object only when the encapsulated computation has completely terminated, rather than as a mere type-converter.

The readers may have noticed that we did not use barriers in Examples 2.13 and 2.15. This is because of the different underlying memory access protocols being used for the objects in those examples. Examples 2.13 and 2.15 use I-structure arrays, while Example 2.16 uses an M-structure array. A barrier may be necessary when the memory access protocol used for implementing an imperative object is not the same as that of the corresponding closed object. We discuss the various memory access protocols below.

Memory Access Protocols

Id defines three classes of data-structures at the language level: Functional, I-structure, and M-structure. Functional data-structures are read-only, I-structures are write-once, and M-structures allow multiple updates. At the architecture level, these data-structures map into the following three kinds of memory access protocols:

Unsynchronized Memory Access — This is the ordinary *load/store* memory access used in conventional architectures. Each memory transaction is assumed to be exclusive and non-blocking. There is no synchronization of any kind between readers and writers.

I-Structure Synchronization — The I-structure protocol [ANP89] enforces producer-consumer synchronization between a single writer and multiple readers using full/empty presence bits on memory locations. A location is deemed empty initially. Multiple readers may issue *I-fetch*es all of which block until the single writer performs an *I-store* changing the state of the location to full. The stored data is then distributed to all the blocked and subsequent readers. Multiple writes to the same location are considered to be an error.

M-Structure Synchronization — The M-structure protocol [BNA91, Bar92] enforces mutual-exclusion synchronization among multiple readers and writers. Readers issue *M-take* operations on full memory locations that read the location and leave it empty. A subsequent *M-put* on the location restores the status to full and makes the data available to other readers. It is possible to allow only one outstanding *M-put* operation and several *M-take*s waiting to succeed as done in Id, or one could queue up both *M-take*s and *M-put*s and match them up.

I-structure and M-structure objects are implemented using their respective memory access protocols, but functional objects may be implemented using either unsynchronized or I-structure access protocol. However, intuitively it should be clear that a given object cannot be accessed using two different protocols simultaneously — that would lead to a run-time error. Therefore, it becomes necessary to ensure that all in-flight imperative operations on an object have terminated before it is closed and accessed as a functional object. A barrier may be inserted just before the `close` operation in order to guarantee this.

Note that we only have to wait for the termination of all memory operations issued from within the scope of the `close` construct because we already ensure that no imperative handle to the object being closed can escape this scope. Of course, no barrier is needed if the underlying memory access protocol remains the same when changing from an imperative to a functional view of the same object. For instance, currently the Id compiler uses the I-structure protocol to implement all functional objects. Therefore, no barrier is needed when closing I-structure objects into functional objects (Examples 2.13 and 2.15), whereas a barrier is required when closing M-structure objects into functional objects that use the I-structure protocol (Example 2.16).

Protocol Conversions

Figure 2.1 depicts all possible protocol conversions at the time of closing an object. An imperative object may be implemented using any one of the three memory access protocols, while a functional object may use either the unsynchronized read protocol or the I-structure read protocol. The arrows depict the protocol conversion implied by the `close` operation. The annotations on the arrows summarize the kind of barrier required, if any, for the underlying protocol conversion. We discuss the various cases below.

When closing an unsynchronized mutable object into an unsynchronized functional object (refer to Figure 2.1), we need to make sure that all previously issued write operations have terminated. Otherwise, the closed object may get updated after being closed. This is enforced by using a *write*-barrier before closing the mutable object.

Although, the Id compiler uses the I-structure protocol to implement all functional objects, it is possible to implement functional objects that are known to be strict without any synchronization. It is also possible to introduce unsynchronized objects as another primitive data class within the language that need not pay the significant overhead of I-structure synchronization, especially when it is emulated in software. In this situation, a *write*-barrier is necessary when closing an I-structure object into an unsynchronized object. Otherwise, subsequent unsynchronized *read* operations would not see the effect of any outstanding *I-store* operations. However,

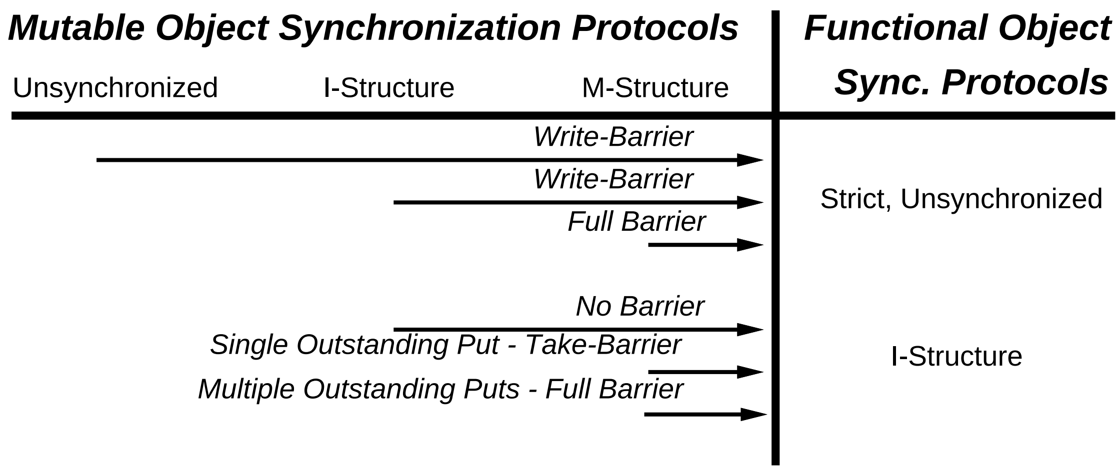

Figure 2.1: Conversions among Synchronization Protocols at the time of Closing.

any outstanding *I-fetch* operations can always be satisfied using the data that is already present, therefore we need not wait for any outstanding *I-fetch* operations to terminate before closing the object.

When closing M-structure objects into unsynchronized functional objects, it is clear that we must wait for both *M-take* and *M-put* operations to terminate before accessing the object with unsynchronized *read* operations. This is because both *M-take* and *M-put* may modify the actual contents of the memory location and all such modifications must complete before it is safe to use the object functionally.

We already mentioned that the I-structure protocol is currently used within the Id compiler to implement both functional and I-structure objects. The only difference between the two at the language-level is that functional objects are allocated and completely defined at the same time and then subsequently used in a read-only fashion, while I-structure objects may be allocated and then independently filled *via* assignment anywhere within the program. Since the underlying synchronization protocol is the same in both cases, no barriers are necessary when closing an I-structure object into a functional object.

Finally, while closing M-structure objects into functional objects that are implemented using the I-structure protocol, if only one outstanding *put* is allowed, then it is possible to use only a *take*-barrier [Bar92] instead of the usual full barrier. This is because once a location is empty after a successful *M-take* operation, multiple functional *I-fetch*es may be allowed to queue up and the ensuing *M-put* can be made to satisfy them just like an *I-store* would.

Discussion

Since there are so many possibilities due to variations in data classes, synchronization protocols and their implementations, henceforth, we shall assume that the `close` construct is always explicitly or implicitly accompanied with the appropriate barrier where necessary. The main thrust of our research is to guarantee type-safety and dynamic non-mutability *via* static analysis which is orthogonal to the issue of guaranteeing dynamic termination of parallel update operations upon closing an object. Therefore, in the rest of this thesis, we will only concentrate upon strict, sequential, unsynchronized accesses to memory as done in most conventional languages.

2.4 Sound Typings for Imperative/Closed Objects

As discussed in the last section, our overall strategy for closing imperative objects can be summarized as follows:

1. First, we have to model the "imperativeness" of objects within the type system.
2. Next, we develop sound verification criteria for the type system under which an object can be safely closed.
3. Finally, we apply the criteria to each object being closed at compile-time, verify the safety of closing and convert the type of the object appropriately if the verification succeeds. Otherwise, we raise a static "close-error".

Following the above outline, in this section we informally discuss the typing machinery required for describing and closing imperative objects and present a set of *closing strategies* under which this operation can be done safely. These strategies form the basis of the formal static and dynamic semantics presented in the next chapter. We also touch upon some language design issue that will be discussed in greater detail in Chapter 4.

2.4.1 Modeling "Imperativeness" in Types

In Section 2.2.7, we motivated our choice of closure typing system of Leroy as a starting point for the typing extensions being proposed in this thesis. We also mentioned that we will need some sort of region-based analysis in order to distinguish among various kinds of imperative and functional objects. In this section, we informally describe this type representation.

Our approach takes a middle ground between the effect and the closure typing system of Leroy. We model the "imperativeness" of an object using *parameterized type constructors* where a simple region expression is attached to each type constructor that identifies whether or not that constructor is imperative. A *region expression* ρ is either a *region variable* r, or the null region ϵ. The intuitive idea is that a type constructor with a null region is considered to be functional, while the presence of a region variable identifies it to be imperative (*c.f.* closure typing system) as well as provides an abstraction for a set of locations associated with that object (*c.f.* effects system with regions). Another way to look at this is that a non-null region expression associated with a type constructor ensures a read/write capability over the objects of that type, while a null region provides a read-only capability over the objects of that type.

As an example, the type of the application `(mkref identity)` (Example 2.3) is shown below under various type systems:[10]

[10] Standard ML notation [MTH90] uses postfix type constructors in type expressions, as in $(u \rightarrow u)$ *ref*. We will follow that notation in Chapter 3 when discussing formal semantics. For now, we use prefix type constructors since they are more intuitive.

Type System	Type of `(mkref identity)`
Standard ML ([MTH90, Tof90])	$ref\ (u \rightarrow u)$
Standard ML/NJ ([AM89, HMV93])	$ref\ (u^0 \rightarrow u^0)$
Simple Effects ([Wri92])	$ref\ (t \xrightarrow{f} t)$ with effect $\{alloc(t, f)\}$
Effects with regions ([TJ92])	$ref\ (t \xrightarrow{f} t)$ with effect $\{alloc_r(t \xrightarrow{f} t)\}$
Closure Type ([Ler92])	$ref\ (t \xrightarrow{u} t)$
Closure Type with regions (this work)	$ref(r)\ (t \xrightarrow{u} t)$

In our representation (the last row), the type constructor *ref* is accompanied by a new unique region variable at every application of `mkref` within the program. These region variables participate in type unification thereby abstractly keeping track of the set of statically aliased reference locations and their scope of accessibility rather than relying on various classes of type variables or a separate set of effects.

The advantage of this representation is that it allows us to close the type of an imperative object by simply replacing the appropriate region variables in a type constructor by the null region ϵ under suitable conditions. The "imperativeness" of an object can still be determined syntactically by examining its parameterized type constructor, so we are still following the closure typing system; no separate effects need to be collected.

Furthermore, a direct correspondence can be established between a user-defined imperative type constructor that is parameterized by one or more non-null regions and a completely functional version of the same type constructor by simply erasing all its qualifying region expressions without disturbing the type constructor itself. For instance, now we can define just a single parameterized array datatype $vector(\rho)$ where a region $\rho = r$ represents an I-structure array and a region $\rho = \epsilon$ represents a functional array.[11] The functional type constructor *vector* is now considered to be a type synonym for $vector(\epsilon)$.

Having independent region variables also separates the issue of type polymorphism from non-mutability quite well. The imperativeness of an object is reflected in the regions associated with its type constructor and not in its polymorphic type variables. Indeed, imperative properties of a monomorphic type such as *point* given below can also be accurately represented and manipulated:

Example 2.17:

```
type point = pt !float !float;
```

The type constructor *point* will be parameterized by two region variables $point(r_1, r_2)$ representing the fact that it has two mutable fields that can be closed independently. The exact association of region variables to mutable fields can be specified explicitly within the type declaration or defined implicitly. We will come back to these language design issues in Chapter 4. Now, let us look at some sound verification strategies for closing imperative objects.

2.4.2 Handling the Environment

Once an imperative object is created and is made accessible as part of the environment at a particular scope, it is nearly impossible to close it safely at that scope or at any scope lexically inside it because many other objects may already hold a write handle to it. That is why we

[11] However, M-structure arrays would still require a separate type constructor in order to distinguish them from I-structure arrays. We will come back to this issue in Chapter 4.

specified the `close` construct as an encapsulator of the entire program fragment that constructs the imperative object (Section 2.3.1) rather than as a mere type-converter. This situation is further complicated by the fact that the scope of accessibility of a mutable object is not always the same as the scope of its allocation because a locally created object may be made accessible non-locally by storing it into a global data-structure. The function `escape_1` of Example 2.15 illustrates this problem. A write handle to the locally allocated object `a` is made accessible by storing it into the global object `b`. Now anybody looking at `b` could get hold of `a` and assign into it. Therefore, it is not safe to close or generalize `a` when it is returned from the function `escape_1`.

Fortunately, modeling the imperativeness of an object using region variables allows us to detect this situation statically. The region variable associated with the type of an imperative object becomes visible in the enclosing type environment when that object is exported into the enclosing value environment. This is illustrated below:

Example 2.18:

```
b = i_vector (1,1);                  % b :: (vector(r1) (vector(r2) t))
def escape_1 n =
  close { a = i_vector (1,n);
          ...
          b[1] = a;                  % a :: (vector(r2) t)
        in a };                      % Unsafe close detected.
```

The assignment (`b[1] = a`) causes the region variable r_2 contained within the type of the array `a` to become visible in the type environment enclosing the `close` construct through the type of array `b`. This fact may be used as a static test while typing the `close` construct to detect such escaping objects. This is summarized in the following typing strategy:

Closing Strategy 1 *An object may be safely closed at the lowest lexical scope higher than the scope of its creation at which none of the region variables contained in its type occur free in the type environment.*

Sometimes, the type of a mutable object can escape into the type environment without actually leaving a write handle around. This may happen if the type of the mutable object is shared with some other global object due to type-unification. This phenomenon is called *region-aliasing* and is illustrated in the following example:

Example 2.19:

```
a = i_vector (1,1);                  % a :: (vector(r1) t)
b = close { c = i_vector (1,2);      % c :: (vector(r2) t)
            d = if ... then a else c;
          in c };                    % Cannot close due to region-aliasing.
```

In the above example, the typing of the conditional expression unifies the region variables r_1 and r_2 of the arrays `a` and `c` respectively. Now, according to Strategy 1, the array `c` cannot be closed because its region variable is visible in the enclosing type environment even though the array itself does not escape into the enclosing scope in any way. Such cases are unavoidable in a conservative, static type inference system.

2.4.3 Handling Structured Results

Until now we were considering cases where a single, flat, local data-structure is closed and returned. The function `escape_2` in Example 2.15 illustrates the case when the mutable object to be closed is returned as part of another object. In general, multiple objects could be closed and returned from a scope and all of them would have to be verified for safety simultaneously because they may refer to each other.

In function escape_2 of Example 2.15, the returned object is a 2-tuple both of whose components point to the same shared array a. Since, the second component of the return tuple provides a write handle to the same array, closing its first component should be illegal. We reproduce the example below:

Example 2.20:
```
def escape_2 n =
  close { a = i_vector (1,n);          % a :: (vector(r) t)
          ...
         in a, a } :: (vector _),_;
```

In terms of types, we observe that the region variable r in the type of array a would get erased in the type of the first component (due to close) but would still be present in the type of the second component. This fact can be used to detect such escaping write handles as expressed in the following typing strategy:

Closing Strategy 2 *Local data returned from a scope is allowed to be closed only if none of the region variables being closed occur free in the remaining type of the returned data.*

The above strategy stresses two important points. First, we must specify exactly which occurrences of region variables we are interested in closing. In some sense, this requires us to specify exactly which fields or locations in a mutable object are we interested in closing.

Second, there should not be any way to access the open version of the object being closed *via* the contents of the object itself. Since the structure of an object is reflected in its type, this check can be performed statically by testing whether any of the region variables being closed are visible in the type of the rest of the object. Note that this does not preclude the possibility of closing recursive or cross-referenced mutable objects. The only restriction is that all references to the same mutable object must be closed simultaneously, otherwise the close operation will not be safe.

Note that the function escape_2 would be acceptable if both the write handles being returned were closed at the same time. The following example would also be acceptable since the region variables associated with the types of a and b are unrelated:

Example 2.21:
```
def escape_2' n =
  close { a = i_vector (1,n);
          b = i_vector (1,n);
         in a, b } :: (vector _),_;
```

Here a may be closed successfully and converted into a functional data-structure while b remains mutable and is typed in the usual way.

2.4.4 Handling Functions

The simple Strategy 2 works well with explicitly nested, first-order data-structures like tuples and arrays. Function closures present a different problem as illustrated by the definition of escape_3 in Example 2.15. Here, a write handle to the array a escapes within the definition of the function `g`. The ordinary Hindley/Milner type of the function can not capture this fact at all since it only records the types of arguments and the result of the function.

This is where Leroy's closure typing information carried on the function type proves useful. Using the closure type, we can easily determine if the region variable being closed is present within the returned closure. If so, then the `close` operation fails. With this addition, the Strategy 2 will be able to detect the escape of the write handle to array `a` from `escape_3` within the closure type of the function `g`.

Note that in the closure typing system, there is no way to distinguish between a function reading from a mutable object and another that writes to it. Therefore, all such functions are conservatively considered to be potential writers and the region variables contained within their closure types should never be closed. This is expressed in the following strategy:

Closing Strategy 3 *Region variables occurring within the closure type of a function are never allowed to be closed.*

In a more expressive effect-based system [TJ92], one might be able to separate functions that only read from a mutable object from those that both read and write the object. In that case, only the latter class is a candidate for potential type-safety violation, the functions that only read from a mutable object may be allowed to close those objects.

2.5 Summary

To summarize, we have informally shown above how to extend a state of the art imperative type system [Ler92] with a type abstraction mechanism that can be used to convert imperative objects into functional objects in a type-safe and transparent manner without the loss of storage efficiency or parallelism. Specifically, we have proposed a new type-domain construct called `close` that controls this type abstraction as a program encapsulator. We have informally shown several typical uses of such a facility, discussed its implications on efficiency, parallelism and dynamic memory access protocols, and outlined possible strategies to verify its correctness within the type system. Finally, we have also given a flavor of the kind of syntactic and semantic machinery that may be required to express, propagate and analyze such information. The next chapter formalizes these ideas in the context of a polymorphic, strict, sequential language and shows a soundness theorem guaranteeing that closed objects verified by our type system cannot be updated during evaluation.

Our guiding principle behind this approach has been to engineer a practically useful notion of encapsulating imperative programs and data-structures into functional abstractions. Our ideas are geared more towards simplicity and run-time performance of user programs (space efficiency and preserving parallelism) rather than towards sheer expressive power of the type system.

Chapter 3

Semantics of "Close"

In this chapter, we describe the semantics of the `close` operation. This semantics is presented in the framework of a small kernel language that supports recursive functions, tuples, and simple reference locations. In Chapter 4, we will extend this system to handle more general data-structures such as arrays and algebraic types. Our type system is a direct extension of the **Closure Typing** system presented in Chapter 3 of Xavier Leroy's Ph.D. thesis [Ler92].

We present the static and the dynamic semantics of our kernel language and show a correspondence between the two in the form of a soundness theorem (Theorem 3.16). This is our main result. It gives us the guarantee that well-typed terms do not run into run-time type-errors. The theorem also implies that mutable objects can be safely considered to be functional once they are successfully closed, *i.e.*, in a type-correct program it is impossible to update an object that has been closed by the type system (Corollaries 3.17 and 3.18). Finally, we use the same type inference algorithm as described in [Ler92] that infers the correct and most general type of every expression in the program.

As far as possible, we have kept the same mathematical notation as used in [Ler92]. Throughout this thesis, all symbols appearing in `typewriter` font are taken verbatim. They denote syntactic entities that stand for themselves. Symbols appearing in SMALL CAPITALS denote classes of objects. Unless specified otherwise, Greek symbols and symbols appearing in *italics* stand for meta-variables that can be replaced with specific object instances in their class.

3.1 Kernel Expression Language

3.1.1 Expression Syntax

The EXPRESSION language is defined below:

EXPRESSIONS:	a	$::=$	c	constant
		$\mid$	x	identifier
		$\mid$	$op(a)$	primitive application
		$\mid$	f `where` $f(x) = a$	recursive function
		$\mid$	$a_1\ a_2$	application
		$\mid$	`let` $x = a_1$ `in` a_2	`let`-binding
		$\mid$	$(a_1, \ldots, a_n)$	n-tuple
		$\mid$	`close` a	`close` expression

In this grammar, x and f range over an infinite set of IDENTIFIERS. c ranges over a predefined set of CONSTANTS including unit (`()`), boolean (`true`, `false`) and integer (...,`-1`,`0`,`1`,...) constants.

In the expression $op(a)$, op ranges over a predefined set of OPERATORS including the usual arithmetic and comparison operators, ith element projection operators for n-tuples, and a ternary conditional operator. This set also includes the primitive operators to allocate (`ref`), dereference (`!`) and assign (`:=`) mutable reference locations that will be described later in more detail. In general, arguments of multi-arity operators are supplied as tuples,[1] but we will freely use special syntax for some common operators, for example (`if`...`then`...`else`...) for the conditional operator, (x`:=`v) for reference assignment, and simple pattern matching for tuple projection.

The expression f `where` $f(x) = a$ denotes user-defined recursive functions. The identifier f can occur inside the expression a. This makes our small language more realistic and allows us to provide meaningful examples. The `let` construct is the source of polymorphism in this language. In some of our Id examples, we represent several `let`-bindings together in a block enclosed within braces ({}). Finally, we have added the `close` construct that enforces functional behavior on the data-structure being returned from the expression a.

The set of FREE IDENTIFIERS of an expression a is denoted by $\mathcal{F}(a)$ and is computed in the usual manner as shown below:

$$\begin{array}{rcl@{\qquad}rcl}
\mathcal{F}(c) & = & \phi & \mathcal{F}(f \texttt{ where } f(x) = a) & = & \mathcal{F}(a) \setminus \{f, x\} \\
\mathcal{F}(x) & = & \{x\} & \mathcal{F}(\texttt{let } x = a_1 \texttt{ in } a_2) & = & \mathcal{F}(a_1) \cup (\mathcal{F}(a_2) \setminus \{x\}) \\
\mathcal{F}(op(a)) & = & \mathcal{F}(a) & \mathcal{F}(a_1, \ldots, a_n) & = & \bigcup_{1 \le i \le n} \mathcal{F}(a_i) \\
\mathcal{F}(a_1\ a_2) & = & \mathcal{F}(a_1) \cup \mathcal{F}(a_2) & \mathcal{F}(\texttt{close } a) & = & \mathcal{F}(a)
\end{array}$$

3.1.2 Dynamic Semantics

The dynamic semantics of the above language is defined using relational semantics. We define a predicate relation between syntactic expressions and results that tells whether a given expression can evaluate to a given result. This relation, called EVALUATION JUDGMENT, is of the following form:

$$e \vdash a/s \Rightarrow r$$

Here e is an ENVIRONMENT, s is an initial STORE, and r is the RESULT of evaluating the expression a under the environment e and the initial store s. Evaluation judgments are established using a system of axioms and inference rules. This technique is also known as "Structured Operational Semantics" (SOS) [Plo81].

Semantic Objects

First, we define the semantic objects used in the dynamic semantics:

[1]Primitive operators are not allowed to be curried.

$$
\begin{array}{llcll}
\text{RESULTS:} & r & ::= & v/s & \text{value and result store} \\
 & & \mid & \texttt{err} & \text{error} \\
\text{VALUES:} & v & ::= & c & \text{constant} \\
 & & \mid & \langle \textsf{n-tup}\ v_1, \ldots, v_n \rangle & n\text{-tuple} \\
 & & \mid & \langle \textsf{clsr}\ f, x, a, e \rangle & \text{function closure} \\
 & & \mid & l & \text{store location} \\
\text{STORABLE VALUES:} & w & ::= & v,\text{rw} & \text{read/write value} \\
 & & \mid & v,\text{ro} & \text{read-only value} \\
\text{ENVIRONMENTS:} & e & ::= & \{x_1 \mapsto v_1, \ldots, x_n \mapsto v_n\} & \\
\text{STORES:} & s & ::= & \{l_1 \mapsto w_1, \ldots, l_n \mapsto w_n\} &
\end{array}
$$

An evaluation can either result in a type-error or it produces a well defined value along with the final store. A well defined value is either a constant base value, a tuple of values, a function closure, or a store location.

Environments bind free identifiers of an expression to values. Stores map locations to storable values that consist of a value and a tag that denotes whether that location has read/write or read-only semantics. This flag is used in defining the semantics of the `close` construct. We assume selector functions $value(w)$ and $tag(w)$ that select the value and tag respectively from a storable value.

Both stores and environments are finite mappings that support the following operations:

Notation 3.1

1. *For any mapping F, we denote the* DOMAIN *of F by $Dom(F)$ and its* RANGE *by $CoDom(F)$.*
2. *The extension of a mapping F at the domain point p with a range value q is written as $F + \{p \mapsto q\}$ and is defined in the usual way:*

$$
(F + \{p \mapsto q\})(x) = \begin{cases} q & \text{if } x = p \\ F(x) & \text{otherwise} \end{cases}
$$

3. *The restriction of a mapping F to the domain A, where $A \subseteq Dom(F)$, is denoted by $F\mid_A$.*
4. *A finite mapping $F = \{p_1 \mapsto q_1, \ldots, p_n \mapsto q_n\}$ is considered to be undefined outside its domain $\{p_1 \cdots p_n\}$ unless specified otherwise.*

Given a value v, we inductively define $\mathcal{L}(v)$ to be the set of all locations directly contained within it:

$$
\begin{array}{rcl}
\mathcal{L}(c) & = & \phi \\
\mathcal{L}(\langle \textsf{n-tup}\ v_1, \ldots, v_n \rangle) & = & \bigcup_{1 \le i \le n} \mathcal{L}(v_i) \\
\mathcal{L}(\langle \textsf{clsr}\ f, x, a, e \rangle) & = & \mathcal{L}(e) \\
\mathcal{L}(l) & = & \{l\}
\end{array}
$$

For an environment $e = \{x_1 \mapsto v_1, \ldots, x_n \mapsto v_n\}$, we define $\mathcal{L}(e) = \bigcup_{1 \le i \le n} \mathcal{L}(v_i)$.

We define the set of locations *reachable* from a given object with respect to a given store as follows:

Definition 3.2 (Reachability) *Given a value v and a store s, we define $Reachable(v, s)$ to be the set of all locations within the domain of s that are either directly contained within v or transitively contained in a value stored at such a location via the store s. This extends naturally* *(pointwise) to values present in an environment e.*

$$
\begin{array}{rcll}
\mathit{Reachable}(c, s) & = & \phi & \\
\mathit{Reachable}(\langle \textsf{n-tup}\ v_1, \ldots, v_n \rangle, s) & = & \bigcup_{1 \le i \le n} \mathit{Reachable}(v_i, s) & \\
\mathit{Reachable}(\langle \textsf{clsr}\ f, x, a, e \rangle, s) & = & \mathit{Reachable}(e, s) & \\
\mathit{Reachable}(l, s) & = & \phi & l \notin \mathit{Dom}(s) \\
\mathit{Reachable}(l, s) & = & \{l\} \cup \mathit{Reachable}(v', s) & \mathit{value}(s(l)) = v' \\
\mathit{Reachable}(e, s) & = & \bigcup_{1 \le i \le n} \mathit{Reachable}(v_i, s) & e = \{x_1 \mapsto v_1, \ldots, x_n \mapsto v_n\}
\end{array}
$$

Although the above definition is correct, it does not lead to a well founded induction on the structure of values because we may have circularly defined data-structures. However, at any given step of evaluation, the size of a value and the number of locations reachable from it are both finite, so we can easily compute the reachable locations using the following recursive algorithm that is guaranteed to terminate:

```
    GATHER-LOCATIONS(v, s, L)
1   case v of
2             c         : return L
3   ⟨n-tup v1, ..., vn⟩ : for i ← 1 to n do
4                             L ← L ∪ GATHER-LOCATIONS(vi, s, L)
5                         return L
6      ⟨clsr f, x, a, e⟩ : let {x1 ↦ v1, ..., xn ↦ vn} = e
7                         for i ← 1 to n do
8                             L ← L ∪ GATHER-LOCATIONS(vi, s, L)
9                         return L
10            l         : if l ∈ L ∨ l ∉ Dom(s) then return L
11                        else let v′ = value(s(l))
12                             return GATHER-LOCATIONS(v′, s, {l} ∪ L)
```

The above algorithm traverses the given value v in a depth-first recursive fashion and accumulates the set of all its reachable locations in the variable L. If the current value is a valid location of the given store, then its contents are recursively traversed at Line 12 only if it is not already in the set L. Thus, no object accessible from the given value is traversed more than once and the algorithm is guaranteed to terminate.

The reachability function given in Definition 3.2 can now be computed as follows:

$$\mathit{Reachable}(v, s) = \textsc{Gather-Locations}(v, s, \phi)$$

Evaluation Rules

Figure 3.1 shows the axioms and inference rules for establishing evaluation judgments $e \vdash a/s \Rightarrow r$. An axiom P allow us to conclude that the proposition P holds. An inference rule is of the form:

$$\frac{P_1 \quad \cdots \quad P_n}{P}$$

All the antecedents $P_1, \ldots, P_n$ must hold in order for us to conclude the consequent P.

The inference rules given in Figure 3.1 provide a strict, sequential, call-by-value semantics for our kernel language. This can be seen from the fact that the store is sequentialized

CONST:	$e \vdash c/s \Rightarrow c/s$
IDENT:	$\dfrac{x \in Dom(e)}{e \vdash x/s \Rightarrow e(x)/s}$
ABS:	$\dfrac{Y = \mathcal{F}(f \;\texttt{where}\; f(x) = a)}{e \vdash (f \;\texttt{where}\; f(x) = a)/s \Rightarrow \langle \textsf{clsr}\; f, x, a, e\mid_Y\rangle/s}$
APP:	$\dfrac{\begin{array}{c} e \vdash a_1/s \Rightarrow \langle \textsf{clsr}\; f, x, a_0, e_0\rangle/s_1 \\ e \vdash a_2/s_1 \Rightarrow v_2/s_2 \\ e_0 + \{f \mapsto \langle \textsf{clsr}\; f, x, a_0, e_0\rangle, x \mapsto v_2\} \vdash a_0/s_2 \Rightarrow v/s_3 \end{array}}{e \vdash (a_1\; a_2)/s \Rightarrow v/s_3}$
TUPLE:	$\dfrac{e \vdash a_1/s \Rightarrow v_1/s_1 \qquad \cdots \qquad e \vdash a_n/s_{n-1} \Rightarrow v_n/s_n}{e \vdash (a_1, \ldots, a_n)/s \Rightarrow \langle \textsf{n-tup}\; v_1, \ldots, v_n\rangle/s_n}$
LET:	$\dfrac{e \vdash a_1/s \Rightarrow v_1/s_1 \qquad e + \{x \mapsto v_1\} \vdash a_2/s_1 \Rightarrow v_2/s_2}{e \vdash (\texttt{let}\; x = a_1 \;\texttt{in}\; a_2)/s \Rightarrow v_2/s_2}$
ALLOC:	$\dfrac{e \vdash a/s \Rightarrow v/s_1 \qquad l \notin Dom(s_1)}{e \vdash \texttt{ref}(a)/s \Rightarrow l/(s_1 + \{l \mapsto v, \mathrm{rw}\})}$
DEREF:	$\dfrac{e \vdash a/s \Rightarrow l/s_1 \qquad l \in Dom(s_1) \qquad value(s_1(l)) = v}{e \vdash \texttt{!}a/s \Rightarrow v/s_1}$
ASSIGN:	$\dfrac{e \vdash a/s \Rightarrow (l, v)/s_1 \qquad l \in Dom(s_1) \qquad tag(s_1(l)) = \mathrm{rw}}{e \vdash \texttt{:=}(a)/s \Rightarrow \texttt{()}/(s_1 + \{l \mapsto v, \mathrm{rw}\})}$
CLOSE:	$\dfrac{\begin{array}{c} e \vdash a/s \Rightarrow l/s_1 \qquad s_1(l) = v, \mathrm{rw} \\ L = Reachable(l, s_1) \cup Reachable(e, s_1) \cup \bigcup_{l' \in Dom(s)} Reachable(l', s_1) \end{array}}{e \vdash (\texttt{close}\; a)/s \Rightarrow l/(s_1\mid_L + \{l \mapsto v, \mathrm{ro}\})}$

Figure 3.1: The Dynamic Semantics of the Kernel Expression Language.

through various computations (APP, TUPLE, and LET rules) and that function and `let` bodies are evaluated in an environment where arguments are bound to values (APP and LET rule).

Figure 3.1 only shows the inference rules that lead to the computation of a well defined value. Our convention for the generation or propagation of the `err` result is as follows. Some rules have antecedents that require pattern matching: the operator in the APP rule must evaluate to a closure value, the expression in the CLOSE rule must evaluate to a location with a read/write tag, the expression in the DEREF rule must evaluate to a location, and the location to be assigned in the ASSIGN rule must have a read/write tag. We add an `err` generating inference rule for every case of mismatch between any of these patterns and the actual values and tags found during their evaluation. Similarly, `err` propagating inference rules are added for each antecedent in an inference rule that may generate an `err` result. In all these cases, the consequent simply evaluates to the `err` result and all propositions following the error generating antecedent are ignored.

Most of the axioms and inference rules shown in Figure 3.1 for the various kernel language constructs are fairly standard and self explanatory. We have shown the primitive operator rules for reference operators only. Usual arithmetic and structural operators (tuple projection) are defined in the usual way. The ALLOC rule initializes new reference locations with a value and a read/write tag. We assume that an infinite set of new locations is available. The DEREF rule reads the value out of an existing location regardless of its tag. The ASSIGN rule only assigns to locations which have a read/write tag.

The CLOSE rule requires a little more explanation. This rule is the only place where the read/write tag of a location is explicitly changed to a read-only tag. This makes that reference object non-mutable. We have also restricted the domain of the final store to the reachable locations of the location being closed, the current environment and the locations of the initial store. This operation removes some non-reachable garbage locations from the final store that may contain references to the location being closed. Although this operation seems somewhat artificial, it is of immense help in reducing the complexity of the soundness proof later on. We motivate the reasons for doing so below.

A more intuitive semantic rule for the `close` construct would be:

$$\text{CLOSE}': \qquad \frac{e \vdash a/s \Rightarrow l/s_1 \qquad s_1(l) = v, \mathrm{rw}}{e \vdash (\texttt{close}\ a)/s \Rightarrow l/(s_1 + \{l \mapsto v, \mathrm{ro}\})}$$

This rule does not restrict the domain of the resulting store. Why would we want to do that operation anyway? The following example brings out the issue:

Example 3.1:

```
a = close {
  b = ref 1;
  c = ref b;
 in b };
```

Within the scope of the `close` block, a freshly allocated reference `c` points to another fresh reference `b`. Both these references are present in the store that is returned from the block although there is no way to access the reference `c` once that block is exited. The unreachable reference to `b` *via* `c` creates technical problems while showing the correspondence between the static and dynamic semantics[2] therefore we would like to get rid of it. One direct way of achieving this is to restrict the domain of the final store to contain just the reachable locations, as we have done in the CLOSE rule above.

The alternate CLOSE′ rule is not wrong. We just have to do more work while showing its soundness restricting our attention to just the reachable locations of the current value and the current environment with respect to the current store at every step of the proof due to the presence of garbage locations such as `c` scattered in its domain. In technical terms, this would imply that all our proofs must be carried out using the method of co-induction (due to the possibility of having cyclic data-structures within the store) rather than a straightforward induction on the structure of the current value and a separate induction involving all the locations in the domain of the current store. Therefore, we have opted for the somewhat non-intuitive CLOSE rule in order to avoid the complex semantic machinery required to show the soundness of the alternate CLOSE′ rule.

[2]Since we have not yet shown the static rule for `close` or the semantic machinery used to show soundness, we request the reader to bear with us for the time being.

3.1.3 Properties of the Evaluation Rules

In order to convert read/write store locations into read-only locations in a safe manner, we need to characterize the allocation, reachability, and manipulation of store locations during an evaluation. In this section, we show two important properties: locations reachable through the result of an evaluation are either new locations or reachable through the evaluation environment (Proposition 3.5), and old locations that get updated during an evaluation are always reachable through the evaluation environment (Proposition 3.6). Both these propositions will be used later in proving the soundness of the `close` construct. But, first we show some auxiliary propositions.

It is evident from the evaluation rules presented in Figure 3.1 that the domain of the store keeps growing during an evaluation. We do not model storage reclamation in these rules. This allows us to state the following:

Proposition 3.3 *let a be an expression, v be a value, e be an environment, and s_0, s_1 be initial and final stores respectively such that $e \vdash a/s_0 \Rightarrow v/s_1$. Then $Dom(s_0) \subseteq Dom(s_1)$.*

Proof: by induction on the length of evaluation derivation for a. A simple examination of the evaluation rules shows that in all cases except the CLOSE rule, either the domain of the store grows or it remains unchanged. In the case of the CLOSE rule, the domain of the final store is possibly smaller than that of the intermediate store due to the domain restriction, but it still includes the entire domain of the initial store by construction. □

Next, we show that a given property applicable to all locations of a store extends inductively to all values and environments that refer to the locations in that store.

Proposition 3.4 *Let e be an environment and s_0, s_1 be stores such that $Dom(s_0) \subseteq Dom(s_1)$, and for all locations $l \in Dom(s_0)$*

$$l' \in [Reachable(l, s_1) \setminus Reachable(l, s_0)] \Longrightarrow l' \notin Dom(s_0) \bigvee l' \in Reachable(e, s_0)$$

Then, for any value v' and environment e' we have,

$$\begin{array}{rcl} l' \in [Reachable(v', s_1) \setminus Reachable(v', s_0)] & \Longrightarrow & l' \notin Dom(s_0) \bigvee l' \in Reachable(e, s_0) \\ l' \in [Reachable(e', s_1) \setminus Reachable(e', s_0)] & \Longrightarrow & l' \notin Dom(s_0) \bigvee l' \in Reachable(e, s_0) \end{array}$$

As a corollary, for $e' = e$ we have,

$$l' \in Reachable(e, s_1) \Longrightarrow l' \notin Dom(s_0) \bigvee l' \in Reachable(e, s_0)$$

Proof: by structural induction on v' and the values contained in the environment e'. We show the various cases for values.

Case 1: v' is c — Trivial, since there are no reachable locations from a constant.

Case 2: v' is $\langle \textsf{n-tup}\ v_1, \ldots, v_n \rangle$ — By definition of reachability for tuples we have,

$$Reachable(\langle \textsf{n-tup}\ v_1, \ldots, v_n \rangle, s) = \bigcup_{1 \le i \le n} Reachable(v_i, s) \tag{3.1}$$

The result follows from above using the induction hypothesis for each individual v_i and the following algebraic identity for arbitrary sets:

$$\left(\bigcup_{1 \le i \le n} X_i \right) \setminus \left(\bigcup_{1 \le i \le n} Y_i \right) \subseteq \bigcup_{1 \le i \le n} (X_i \setminus Y_i) \tag{3.2}$$

Case 3: v' is $\langle \mathsf{clsr}\ f, x, a_0, e_0 \rangle$ — Same as above.

Case 4: v' is l — If $l \notin \mathit{Dom}(s_1)$ or $l \notin \mathit{Dom}(s_0)$ then we have nothing to prove. Otherwise the result follows from the given relation regarding locations.

The environment hypothesis follows from the value hypothesis using the definition of reachability for environments and Equation 3.2. □

Now we prove the proposition that partitions the locations reachable from the result of a evaluation into those that are freshly allocated and those that are reachable from the evaluation environment.

Proposition 3.5 (Fresh Locations) *Let a be an expression, v be a value, e be an environment, and s_0, s_1 be initial and final stores respectively such that $e \vdash a/s_0 \Rightarrow v/s_1$. Then,*

$$l' \in \mathit{Reachable}(v, s_1) \Longrightarrow l' \notin \mathit{Dom}(s_0) \bigvee l' \in \mathit{Reachable}(e, s_0)$$

and for all locations $l \in \mathit{Dom}(s_0)$,

$$l' \in [\mathit{Reachable}(l, s_1) \setminus \mathit{Reachable}(l, s_0)] \Longrightarrow l' \notin \mathit{Dom}(s_0) \bigvee l' \in \mathit{Reachable}(e, s_0)$$

Proof: by induction on the length of evaluation derivation for a. We consider the various cases for the last evaluation rule in the derivation.

Case 1: CONST — Trivial, since $\mathit{Reachable}(c, s_1) = \phi$ and $s_0 = s_1$.

Case 2: IDENT — Trivial, since $\mathit{Reachable}(e(x), s_1) \subseteq \mathit{Reachable}(e, s_1)$ and $s_0 = s_1$.

Case 3: ABS — Trivial, since $\mathit{Reachable}(\langle \mathsf{clsr}\ f, x, a, e\ |_Y \rangle, s_1) = \mathit{Reachable}(e\ |_Y, s_1) \subseteq \mathit{Reachable}(e, s_1)$ and $s_0 = s_1$.

Case 4: APP — The evaluation rule is:

$$\frac{\begin{array}{c} e \vdash a_1/s \Rightarrow \langle \mathsf{clsr}\ f, x, a_0, e_0 \rangle / s_1 \\ e \vdash a_2/s_1 \Rightarrow v_2/s_2 \\ e_0 + \{f \mapsto \langle \mathsf{clsr}\ f, x, a_0, e_0 \rangle, x \mapsto v_2\} \vdash a_0/s_2 \Rightarrow v/s_3 \end{array}}{e \vdash (a_1\ a_2)/s \Rightarrow v/s_3}$$

Let $e_1 = e_0 + \{f \mapsto \langle \mathsf{clsr}\ f, x, a_0, e_0 \rangle, x \mapsto v_2\}$. First, we show the value hypothesis for this case, *i.e.*, we show:

$$l' \in \mathit{Reachable}(v, s_3) \Longrightarrow l' \notin \mathit{Dom}(s) \bigvee l' \in \mathit{Reachable}(e, s) \tag{3.3}$$

Applying the induction hypothesis for values to the last premise we obtain:

$$l' \in \mathit{Reachable}(v, s_3) \Longrightarrow l' \notin \mathit{Dom}(s_2) \bigvee l' \in \mathit{Reachable}(e_1, s_2) \tag{3.4}$$

Note that $l' \notin \mathit{Dom}(s_2)$ implies $l' \notin \mathit{Dom}(s)$ because $\mathit{Dom}(s) \subseteq \mathit{Dom}(s_2)$ from Proposition 3.3. If $l' \in \mathit{Reachable}(e_1, s_2)$, then using the definition of reachability and e_1 we have the following two cases:

- $l' \in \mathit{Reachable}(e_0, s_2)$ — In this case, we use the induction hypothesis for locations on the second premise in Proposition 3.4 with environment $e' = e_0$ to obtain:

$$l' \in [\mathit{Reachable}(e_0, s_2) \setminus \mathit{Reachable}(e_0, s_1)] \Longrightarrow l' \notin \mathit{Dom}(s_1) \bigvee l' \in \mathit{Reachable}(e, s_1) \tag{3.5}$$

To eliminate $\mathit{Reachable}(e_0, s_1)$, we use the induction hypothesis for values on the first premise to obtain:

$$l' \in \mathit{Reachable}(e_0, s_1) \Longrightarrow l' \notin \mathit{Dom}(s) \bigvee l' \in \mathit{Reachable}(e, s) \tag{3.6}$$

Also, we simplify $l' \in \mathit{Reachable}(e, s_1)$ on the right hand side of Equation 3.5 by applying the corollary in Proposition 3.4 for the first premise:

$$l' \in \mathit{Reachable}(e, s_1) \Longrightarrow l' \notin \mathit{Dom}(s) \bigvee l' \in \mathit{Reachable}(e, s) \tag{3.7}$$

Combining Equations 3.5, 3.6, and 3.7 we obtain the following as desired:

$$l' \in \mathit{Reachable}(e_0, s_2) \Longrightarrow l' \notin \mathit{Dom}(s) \bigvee l' \in \mathit{Reachable}(e, s) \tag{3.8}$$

- $l' \in \mathit{Reachable}(v_2, s_2)$ — In this case, we use the induction hypothesis for values on the second premise and then simplify as above using Proposition 3.4 to obtain,

$$\begin{aligned} l' \in \mathit{Reachable}(v_2, s_2) &\Longrightarrow l' \notin \mathit{Dom}(s_1) \bigvee l' \in \mathit{Reachable}(e, s_1) \\ &\Longrightarrow l' \notin \mathit{Dom}(s) \bigvee l' \in \mathit{Reachable}(e, s) \end{aligned} \tag{3.9}$$

Combining statements 3.8 and 3.9 proves the statement 3.3 as desired.

Now we show the location hypothesis, *i.e.*, for all locations $l \in \mathit{Dom}(s)$ we show that:

$$l' \in [\mathit{Reachable}(l, s_3) \setminus \mathit{Reachable}(l, s)] \Longrightarrow l' \notin \mathit{Dom}(s) \bigvee l' \in \mathit{Reachable}(e, s) \tag{3.10}$$

We use the following algebraic identity that is true for arbitrary sets:

$$X \setminus Y \subseteq (X \setminus Z) \cup (Z \setminus Y) \tag{3.11}$$

Using this identity, we obtain:

$$\begin{aligned} &[\mathit{Reachable}(l, s_3) \setminus \mathit{Reachable}(l, s)] \\ &\subseteq [\mathit{Reachable}(l, s_3) \setminus \mathit{Reachable}(l, s_2)] \cup [\mathit{Reachable}(l, s_2) \setminus \mathit{Reachable}(l, s)] \\ &\subseteq [\mathit{Reachable}(l, s_3) \setminus \mathit{Reachable}(l, s_2)] \cup [\mathit{Reachable}(l, s_2) \setminus \mathit{Reachable}(l, s_1)] \cup \\ &\quad [\mathit{Reachable}(l, s_1) \setminus \mathit{Reachable}(l, s)] \end{aligned} \tag{3.12}$$

Now we use the induction hypothesis for locations for each of the three clauses on the right and simplify using Propositions 3.4 and 3.3 to obtain the desired result.

Case 5: TUPLE — The location hypothesis is shown exactly like the case above. We give the argument for the value hypothesis. The evaluation rule is:

$$\frac{e \vdash a_1/s_0 \Rightarrow v_1/s_1 \qquad \cdots \qquad e \vdash a_n/s_{n-1} \Rightarrow v_n/s_n}{e \vdash (a_1, \ldots, a_n)/s_0 \Rightarrow \langle \textsf{n-tup}\ v_1, \ldots, v_n \rangle / s_n}$$

We have to show that:

$$l' \in \mathit{Reachable}(\langle \textsf{n-tup}\ v_1, \ldots, v_n \rangle, s_n) \Longrightarrow l' \notin \mathit{Dom}(s_0) \bigvee l' \in \mathit{Reachable}(e, s_0) \tag{3.13}$$

Applying the induction hypothesis for values to each premise ($1 \leq i \leq n$) and simplifying using Propositions 3.3 and 3.4 we obtain:

$$l' \in \mathit{Reachable}(v_i, s_i) \Longrightarrow l' \notin \mathit{Dom}(s_0) \bigvee l' \in \mathit{Reachable}(e, s_0) \tag{3.14}$$

In order to show Equation 3.13, we need to strengthen Equation 3.14 to $l' \in \mathit{Reachable}(v_i, s_n)$ $(1 \leq i \leq n)$. We use the algebraic identity 3.11 repeatedly to obtain the following:

$$[\mathit{Reachable}(v_i, s_n) \backslash \mathit{Reachable}(v_i, s_i)] \subseteq \bigcup_{n \geq j > i} [\mathit{Reachable}(v_i, s_j) \backslash \mathit{Reachable}(v_i, s_{j-1})] \quad (3.15)$$

We use the induction hypothesis for locations and Proposition 3.4 to simplify each of the clauses on the right in the above statement and plug in Equation 3.14 to obtain the desired result of Equation 3.13.

Case 6: LET — Same argument as in the APP case.

Case 7: ALLOC — The result follows from the induction hypothesis and the fact that the allocated location is in fact chosen to be a new location that is not present in $\mathit{Dom}(s_1)$ and hence not present in $\mathit{Dom}(s)$.

Case 8: DEREF — The result follows directly from the induction hypothesis and the definition of reachability for locations.

Case 9: ASSIGN — The evaluation rule is:

$$\frac{e \vdash a/s \Rightarrow (l, v)/s_1 \qquad l \in \mathit{Dom}(s_1) \qquad \mathit{tag}(s_1(l)) = \mathrm{rw}}{e \vdash \texttt{:=}(a)/s \Rightarrow \texttt{()}/(s_1 + \{l \mapsto v, \mathrm{rw}\})}$$

The value hypothesis follows immediately since no locations are reachable from `()`. For the location hypothesis, note that the final store $s_2 = s_1 + \{l \mapsto v, \mathrm{rw}\}$ differs from the intermediate store s_1 only at location l. Furthermore, using the induction hypothesis for values we know that:

$$l' \in \mathit{Reachable}(v, s_1) \Longrightarrow l' \notin \mathit{Dom}(s) \bigvee l' \in \mathit{Reachable}(e, s) \quad (3.16)$$

Thus, the location hypothesis will be valid for the location l as well which is assigned the new value v.

Case 10: CLOSE — By construction, the final store contains all the reachable locations from the location being closed, the current environment, and the old store. Thus, both value and location hypotheses follow directly from the induction hypothesis since changing the tag of a location does not affect its reachability.

□

Finally, we show the proposition that characterizes the set of locations that may get updated during an evaluation.

Proposition 3.6 (Updated Locations) *Let a be an expression, v be a value, e be an environment, and s_0, s_1 be initial and final stores respectively such that $e \vdash a/s_0 \Rightarrow v/s_1$. Then for any location $l \in \mathit{Dom}(s_0)$,*

$$\mathit{value}(s_0(l)) \neq \mathit{value}(s_1(l)) \Longrightarrow l \in \mathit{Reachable}(e, s_0)$$

That is, pre-existing locations that get updated during an evaluation are reachable from the environment.

Proof: by induction on the length of evaluation derivation for a. We consider the various cases for the last evaluation rule in the derivation.

Case 1: CONST, IDENT, and ABS — Trivial, since $s_0 = s_1$.

Case 2: APP — The evaluation rule is:

$$\frac{\begin{array}{c} e \vdash a_1/s \Rightarrow \langle \mathsf{clsr}\ f, x, a_0, e_0\rangle/s_1 \\ e \vdash a_2/s_1 \Rightarrow v_2/s_2 \\ e_0 + \{f \mapsto \langle \mathsf{clsr}\ f, x, a_0, e_0\rangle, x \mapsto v_2\} \vdash a_0/s_2 \Rightarrow v/s_3 \end{array}}{e \vdash (a_1\ a_2)/s \Rightarrow v/s_3}$$

Three possibilities arise for $value(s(l)) \neq value(s_3(l))$:

1. $value(s(l)) \neq value(s_1(l))$ — The result follows immediately by applying the induction hypothesis to the first premise.
2. $value(s(l)) = value(s_1(l))$ but $value(s_1(l)) \neq value(s_2(l))$ — Using the induction hypothesis on the second premise we obtain that $l \in Reachable(e, s_1)$. Using Proposition 3.5 together with Proposition 3.4 for environments we obtain that

$$l \in Reachable(e, s_1) \Longrightarrow l \notin Dom(s) \bigvee l \in Reachable(e, s) \tag{3.17}$$

   Since we know that $l \in Dom(s)$, the result follows.
3. $value(s(l)) = value(s_1(l)) = value(s_2(l))$ but $value(s_2(l)) \neq value(s_3(l))$ — Using induction hypothesis on the third premise we obtain that $l \in Reachable(e_1, s_2)$ where $e_1 = e_0 + \{f \mapsto \langle \mathsf{clsr}\ f, x, a_0, e_0\rangle, x \mapsto v_2\}$. This can be simplified to the desired result just as in the proof of Proposition 3.5.

Case 3: TUPLE and LET — Same argument as above.

Case 4: ALLOC and DEREF — The result follows directly from induction hypothesis.

Case 5: ASSIGN — The evaluation rule is:

$$\frac{e \vdash a/s \Rightarrow (l, v)/s_1 \qquad l \in Dom(s_1) \qquad tag(s_1(l)) = \mathrm{rw}}{e \vdash \mathtt{:=}(a)/s \Rightarrow ()/(s_1 + \{l \mapsto v, \mathrm{rw}\})}$$

For all locations other than l, the result follows from the induction hypothesis. In case of location l, we apply Proposition 3.5 to the first premise and obtain that,

$$l' \in Reachable((l, v), s_1) \Longrightarrow l' \notin Dom(s) \bigvee l' \in Reachable(e, s) \tag{3.18}$$

It is clear that l is reachable from the pair (l, v). The result follows from the above statement and the induction hypothesis that $l \in Dom(s)$.

Case 6: CLOSE — All locations reachable from the initial store s are included in the final store by construction. Furthermore, the values present at these locations are the same as those in the store s_1. Thus, the result follows from induction hypothesis.

□

3.2 A Closure Typing System

Now we will describe our extension to Xavier Leroy's closure typing system [Ler92].

3.2.1 Type Syntax

The type grammar is defined below:

$$
\begin{array}{llcll}
\text{TYPE VARIABLES:} & \alpha, \beta & ::= & t & \text{regular type variable} \\
& & \mid & u & \text{closure extension variable} \\
& & \mid & r & \text{region variable} \\
\text{TYPES:} & \tau & ::= & t & \text{regular type variable} \\
& & \mid & \iota & \text{base type} \\
& & \mid & \tau_1 \prec\!\langle \pi \rangle\!\succ\!\rightarrow \tau_2 & \text{function type} \\
& & \mid & \tau_1, \ldots, \tau_n & n\text{-tuple type} \\
& & \mid & \tau\ \mathit{ref}(r) & \text{mutable reference type} \\
& & \mid & \tau\ \mathit{ref}(\epsilon) & \text{non-mutable reference type} \\
\text{CLOSURE TYPES:} & \pi & ::= & u & \text{closure extension variable} \\
& & \mid & \sigma, \pi & \text{closure type} \\
\text{REGIONS:} & \rho & ::= & r & \text{region variable} \\
& & \mid & \epsilon & \text{null region} \\
\text{TYPE SCHEMES:} & \sigma & ::= & \forall \alpha_1 \ldots \alpha_n.\, \tau &
\end{array}
$$

In this grammar, a function type ($\rightarrow$) is decorated with a CLOSURE TYPE which is a set of type schemes together with a closure extension variable u. The closure type of a function corresponds to the type schemes of the free identifiers of the function that are stored in its closure environment. The order of occurrence of the type schemes in a closure type does not matter. Note that the above grammar does not allow more than one closure extension variable in a closure type.

A reference type is parameterized by a REGION expression which could be a region variable r or the null region constant ϵ. Regions serve to model the mutability of store locations, while types serve to model the structure of dynamic values. That is why the domain of regions is much simpler than the domain of types.

A region variable parameter r on a mutable reference type serves two purposes. It identifies the reference type as being mutable and it also serves as an abstract static label for the corresponding dynamic mutable location (and any other locations aliased to it) that has that type. This abstraction is useful in tracking the dynamic mutable locations reachable from a given object by statically observing the region variables present within its type. We will formalize this correspondence between regions variables and mutable locations in Section 3.3. Non-mutable or "closed" references are identified by a fixed null region constant (ϵ) because there is no need to keep track of locations that have been closed. Note that $\mathit{ref}(r)$ and $\mathit{ref}(\epsilon)$ are considered to be distinct type constructors; they have a similar form only for syntactic uniformity.

For any type object T, where T may be a type, a closure type, a region, or a type scheme, its FREE VARIABLES $\mathcal{F}(T)$ are defined inductively as follows:[3]

$$
\begin{array}{rcl@{\qquad}rcl}
\mathcal{F}(t) & = & \{t\} & \mathcal{F}(u) & = & \{u\} \\
\mathcal{F}(\iota) & = & \phi & \mathcal{F}(\sigma, \pi) & = & \mathcal{F}(\sigma) \cup \mathcal{F}(\pi) \\
\mathcal{F}(\tau_1 \prec\!\langle \pi \rangle\!\succ\!\rightarrow \tau_2) & = & \mathcal{F}(\tau_1) \cup \mathcal{F}(\pi) \cup \mathcal{F}(\tau_2) & \mathcal{F}(r) & = & \{r\} \\
\mathcal{F}(\tau_1, \ldots, \tau_n) & = & \bigcup_{1 \leq i \leq n} \mathcal{F}(\tau_i) & \mathcal{F}(\epsilon) & = & \phi \\
\mathcal{F}(\tau\ \mathit{ref}(\rho)) & = & \mathcal{F}(\tau) \cup \mathcal{F}(\rho) & \mathcal{F}(\forall \alpha_1 \ldots \alpha_n.\, \tau) & = & \mathcal{F}(\tau) \setminus \{\alpha_1 \ldots \alpha_n\}
\end{array}
$$

In a type scheme $\sigma = \forall \alpha_1 \ldots \alpha_n.\, \tau$, the variables $\{\alpha_1 \ldots \alpha_n\}$ are called the BOUND VARIABLES denoted by $\mathcal{B}(\sigma)$. For any type object T, we also define the DANGEROUS VARIABLES $\mathcal{D}(T)$ and the DANGEROUS REGION VARIABLES $\mathcal{R}(T)$ inductively as follows:

[3]Note that we are using the same notation here as that for computing the free identifiers of an expression because it represents the same concept. The meaning is always clear by context since we never mix types and expressions.

$$
\begin{array}{rcl}
\mathcal{D}(t) & = & \phi \\
\mathcal{D}(\iota) & = & \phi \\
\mathcal{D}(\tau_1 \overset{\langle\pi\rangle}{\rightarrow} \tau_2) & = & \mathcal{D}(\pi) \\
\mathcal{D}(\tau_1, \ldots, \tau_n) & = & \bigcup_{1 \le i \le n} \mathcal{D}(\tau_i) \\
\mathcal{D}(\tau \ ref(r)) & = & \mathcal{F}(\tau \ ref(r)) \\
\mathcal{D}(\tau \ ref(\epsilon)) & = & \mathcal{D}(\tau)
\end{array}
\qquad
\begin{array}{rcl}
\mathcal{D}(u) & = & \phi \\
\mathcal{D}(\sigma, \pi) & = & \mathcal{D}(\sigma) \cup \mathcal{D}(\pi) \\
\mathcal{D}(r) & = & \phi \\
\mathcal{D}(\epsilon) & = & \phi \\
\mathcal{D}(\forall \alpha_1 \ldots \alpha_n . \, \tau) & = & \mathcal{D}(\tau) \setminus \{\alpha_1 \ldots \alpha_n\}
\end{array}
$$

$$
\begin{array}{rcl}
\mathcal{R}(t) & = & \phi \\
\mathcal{R}(\iota) & = & \phi \\
\mathcal{R}(\tau_1 \overset{\langle\pi\rangle}{\rightarrow} \tau_2) & = & \mathcal{R}(\pi) \\
\mathcal{R}(\tau_1, \ldots, \tau_n) & = & \bigcup_{1 \le i \le n} \mathcal{R}(\tau_i) \\
\mathcal{R}(\tau \ ref(\rho)) & = & \mathcal{R}(\tau) \cup \mathcal{F}(\rho)
\end{array}
\qquad
\begin{array}{rcl}
\mathcal{R}(u) & = & \phi \\
\mathcal{R}(\sigma, \pi) & = & \mathcal{R}(\sigma) \cup \mathcal{R}(\pi) \\
\mathcal{R}(r) & = & \phi \\
\mathcal{R}(\epsilon) & = & \phi \\
\mathcal{R}(\forall \alpha_1 \ldots \alpha_n . \, \tau) & = & \mathcal{R}(\tau) \setminus \{\alpha_1 \ldots \alpha_n\}
\end{array}
$$

Specifically, for a mutable reference, the region associated with that type and all type variables contained within it are considered to be dangerous. The variables occurring inside a non-mutable reference type are not considered to be dangerous. For a function closure, the typing rules shown later ensure that the types of all objects reachable from the closure environment are recorded in its closure type. Therefore, the types of mutable references accessible *via* the closure environment are also visible in its closure type and are considered to be dangerous.

Using the type abstractions shown above, we can accurately capture and control the static (type polymorphism) and the dynamic (mutability) properties of imperative data-structures. The basic idea of our type system is to use the type of a composite object as a clue to the reachable mutable reference locations contained within it. Dangerous variables provide this clue directly from the overall type of an object. Intuitively, dangerous type variables model the polymorphic values stored within mutable objects and the dangerous region variables model the mutable locations contained within such objects.

3.2.2 Static Semantics

The static semantics of our kernel language is defined in the same manner as its dynamic semantics. We define a predicate relation between syntactic expressions and types that tells that a given expression elaborates to a given type. This relation, called ELABORATION JUDGMENT, is of the following form:

$$E \vdash a : \tau$$

Here E is a TYPE ENVIRONMENT which is defined below as a finite mapping from identifiers to type schemes.

TYPE ENVIRONMENTS: $\qquad E \ ::= \ \{x_1 \mapsto \sigma_1, \ldots, x_n \mapsto \sigma_n\}$

TYPE SUBSTITUTIONS over this type algebra are finite mappings from regular type variables to types, from closure extension variables to closure types, and from region variables to other region variables. We do not allow substituting region variables with the null region (ϵ) because that would convert a mutable reference type into a non-mutable reference type. This operation should only be performed when it is determined to be safe and is explicitly done using the `close` construct.

TYPE SUBSTITUTIONS: $\qquad \psi, \varphi \ ::= \ \{t \mapsto \tau, \ldots, u \mapsto \pi, \ldots, r \mapsto r', \ldots\}$

Type substitutions are taken to be the identity mapping outside their specified finite domain. They also extend naturally over types, closure types, and type schemes, being applied to their free variables in each case. For a type scheme $\sigma = \forall\alpha_1 \ldots \alpha_n.\ \tau$, it may be necessary to rename some its bound variables α_i so that they are OUT OF REACH for the type substitution φ, *i.e.*, no α_i is in $Dom(\varphi)$ and no α_i occurs free in any type, closure type or region in $CoDom(\varphi)$. Then, the substitution is defined by:

$$\varphi(\forall\alpha_1 \ldots \alpha_n.\ \tau) = \forall\alpha_1 \ldots \alpha_n.\ \varphi(\tau)$$

The INSTANTIATION of a type scheme $\sigma = \forall\alpha_1 \ldots \alpha_n.\ \tau_0$ to a type τ, written as $\sigma \geq \tau$, is defined if there exists a type substitution φ with $Dom(\varphi) \subseteq \{\alpha_1 \ldots \alpha_n\}$ such that $\tau = \varphi(\tau_0)$.

In order to simplify our notation for computing free and dangerous variables of sets of objects, we use the following convention:

Notation 3.7 *Given a set of objects P,*

1. $\bigcup_{p \in P} \varphi(p) = \varphi(P)$.
2. $\bigcup_{p \in P} \mathcal{F}(p) = \mathcal{F}(P)$.
3. $\bigcup_{p \in P} \mathcal{D}(p) = \mathcal{D}(P)$.

The effect of type substitutions on the free and dangerous variables is now captured in the following proposition:

Proposition 3.8 *Let φ be a type substitution. For any T, where T could be a type τ, a closure type π, a region ρ, or a type scheme σ, we have:*

$$\begin{array}{rcccl} & & \mathcal{F}(\varphi(T)) & = & \mathcal{F}(\varphi(\mathcal{F}(T))) \\ \mathcal{F}(\varphi(\mathcal{D}(T))) & \subseteq & \mathcal{D}(\varphi(T)) & \subseteq & \mathcal{F}(\varphi(\mathcal{D}(T))) \cup \mathcal{D}(\varphi(\mathcal{F}(T))) \end{array}$$

Proof: Both these relations follow directly from the definitions of $\mathcal{F}(T)$ and $\mathcal{D}(T)$ by a simultaneous structural induction over the appropriate type object T. □

The first equation provides an exact relationship between the free variables of a type before and after applying a type substitution to it. On the other hand, the second pair of inequalities provide only an approximation to the set of dangerous variables of a type after applying a type substitution to it. This is so because the substitution images of dangerous variables of a type $(\mathcal{F}(\varphi(\mathcal{D}(\tau))))$ may not cover all the dangerous variables of the substituted type $(\mathcal{D}(\varphi(\tau)))$. Some non-dangerous variable may get substituted with a type containing dangerous variables that must also be counted as dangerous in the final type.

Typing Rules

Figure 3.2 shows the axioms and the inference rules for establishing elaboration judgments $E \vdash a : \tau$. The CONST and the PRIMAPP rules establish the elaboration judgment for a constant or a primitive operator application according to a predefined relation *typeof* that provides the type scheme associated with them. All such predefined type schemes are fully-quantified: there are no free variables in these type schemes. Most constants and operators have the obvious type schemes. We only show the predefined type schemes of the three reference operators below:

$$\begin{array}{rcl} typeof(\texttt{ref}) & = & \forall t, u, r.\ t \multimap\!\langle u \rangle\!\!\rightarrow t\ ref(r) \\ typeof(\texttt{!}_{mutable}) & = & \forall t, u, r.\ t\ ref(r) \multimap\!\langle u \rangle\!\!\rightarrow t \\ typeof(\texttt{!}_{non\text{-}mutable}) & = & \forall t, u.\ t\ ref(\epsilon) \multimap\!\langle u \rangle\!\!\rightarrow t \\ typeof(\texttt{:=}) & = & \forall t, u, r.\ (t\ ref(r), t) \multimap\!\langle u \rangle\!\!\rightarrow unit \end{array}$$

$$\text{CONST:}\quad \frac{\mathit{typeof}(c) \geq \tau}{E \vdash c : \tau}$$

$$\text{PRIMAPP:}\quad \frac{\mathit{typeof}(op) \geq \tau_1 \prec\!\langle\pi\rangle\!\!\rightarrow \tau_2 \qquad E \vdash a : \tau_1}{E \vdash op(a) : \tau_2}$$

$$\text{IDENT:}\quad \frac{x \in \mathit{Dom}(E) \qquad E(x) \geq \tau}{E \vdash x : \tau}$$

$$\text{ABS:}\quad \frac{\begin{array}{c}\{y_1 \ldots y_n\} = \mathcal{F}(f \texttt{ where } f(x) = a) \\ E + \{f \mapsto \tau_1 \prec\!\langle E(y_1), \ldots, E(y_n), \pi\rangle\!\!\rightarrow \tau_2, x \mapsto \tau_1\} \vdash a : \tau_2\end{array}}{E \vdash (f \texttt{ where } f(x) = a) : \tau_1 \prec\!\langle E(y_1), \ldots, E(y_n), \pi\rangle\!\!\rightarrow \tau_2}$$

$$\text{APP:}\quad \frac{E \vdash a_1 : \tau_1 \prec\!\langle\pi\rangle\!\!\rightarrow \tau_2 \qquad E \vdash a_2 : \tau_1}{E \vdash a_1\ a_2 : \tau_2}$$

$$\text{TUPLE:}\quad \frac{E \vdash a_1 : \tau_1 \qquad \cdots \qquad E \vdash a_n : \tau_n}{E \vdash a_1, \ldots, a_n : \tau_1, \ldots, \tau_n}$$

$$\text{LET:}\quad \frac{E \vdash a_1 : \tau_1 \qquad E + \{x \mapsto \mathit{Gen}(E, \tau_1)\} \vdash a_2 : \tau_2}{E \vdash (\texttt{let } x = a_1 \texttt{ in } a_2) : \tau_2}$$

$$\text{CLOSE:}\quad \frac{E \vdash a : \tau\ \mathit{ref}(r) \qquad r \notin (\mathcal{F}(E) \cup \mathcal{F}(\tau))}{E \vdash (\texttt{close } a) : \tau\ \mathit{ref}(\epsilon)}$$

Figure 3.2: The Static Semantics of the Kernel Expression Language.

There are two different types for the dereference operator (`!`), one for mutable references and the other for non-mutable references. This is because we consider mutable reference types as distinct from non-mutable reference types. Essentially, we *overload* the use of the dereference operation with these two types. This does not create any problem since the exact type to be used is always clear from context. Moreover, in our kernel language, the underlying dynamic dereferencing operation is the same in both cases.

The IDENT rule *instantiates* the type scheme of an identifier stored in the type environment. The ABS rule shows how closure types are created in this system. The type schemes of all the free identifiers of the function are stored in its closure type. This is necessary to keep track of the mutable locations accessible through the closure environment. The APP and the TUPLE rules are self explanatory. The APP rule also handles primitive operator applications.

The LET rule allows a type to be quantified and added to the type environment as a type scheme. The GENERALIZATION operation in the LET rule is defined as follows:

$$\mathit{Gen}(E, \tau) = \forall \alpha_1 \ldots \alpha_n .\, \tau \quad \text{where} \quad \{\alpha_1 \ldots \alpha_n\} = \mathcal{F}(\tau) \setminus \mathcal{D}(\tau) \setminus \mathcal{F}(E)$$

Finally, the CLOSE rule converts a mutable reference type into a non-mutable reference type by *erasing* its region variable and replacing it with the null region (ϵ). This is an explicit type conversion operation on the mutable reference type. The side condition ensures the soundness of this operation by checking that the region being closed does not escape from the current scope.

This is the exact formalization of the informal closing strategies described in Section 2.4.

3.2.3 Properties of the Typing Rules

In this section, we will present some syntactic properties of the typing rules presented above. The most important property is the following proposition that states that typing is stable under type substitution. This property is essential for performing type inference (Section 3.4) because it guarantees that all incremental type refinements (*via* type substitutions) to a given typing of an expression yield legal typings of that expression. Thus, the typing of an expression can be automatically refined to match that of its enclosing context.

Proposition 3.9 (Stability under Type Substitution) *Let a be an expression, τ be a type, E be a type environment, and φ be a substitution. If $E \vdash a : \tau$, then $\varphi(E) \vdash a : \varphi(\tau)$.*

Proof: by structural induction over a. For completeness, we show all the cases.

Case 1: a is c — The CONST rule applies:

$$\frac{\mathit{typeof}(c) \geq \tau}{E \vdash c : \tau}$$

Let $\mathit{typeof}(c) = \forall \alpha_1 \ldots \alpha_n .\ \tau_0$ and ψ be its instantiation substitution such that $\tau = \psi(\tau_0)$ with $\mathit{Dom}(\psi) \subseteq \{\alpha_1 \ldots \alpha_n\}$. After renaming if necessary, assume that α_i are out of reach for φ. Now define a substitution ψ' with domain $\{\alpha_1 \ldots \alpha_n\}$ such that $\psi'(\alpha_i) = \varphi(\psi(\alpha_i))$.

Since the type scheme $\mathit{typeof}(c)$ is assumed to be fully-quantified, there are no free variables in τ_0 other than the α_i. Thus $\psi'(\tau_0) = \varphi(\psi(\tau_0)) = \varphi(\tau)$, which implies that $\mathit{typeof}(c) \geq \varphi(\tau)$. The desired result follows using the CONST rule.

Case 2: a is $op(a)$ — We proceed exactly like the previous case to show that $\mathit{typeof}(op) \geq \varphi(\tau_1 \prec\!\pi\!\succ\!\!\rightarrow \tau_2)$. The desired result follows from the induction hypothesis on the second antecedent.

Case 3: a is x — The IDENT rule applies:

$$\frac{x \in \mathit{Dom}(E) \qquad E(x) \geq \tau}{E \vdash x : \tau}$$

Let $E(x) = \forall \alpha_1 \ldots \alpha_n .\ \tau_0$ and ψ be its instantiation substitution such that $\tau = \psi(\tau_0)$ with $\mathit{Dom}(\psi) \subseteq \{\alpha_1 \ldots \alpha_n\}$. After renaming if necessary, assume that α_i are out of reach for φ, so that $\varphi(E(x)) = \forall \alpha_1 \ldots \alpha_n .\ \varphi(\tau_0)$. Now define a substitution ψ' with domain $\{\alpha_1 \ldots \alpha_n\}$ such that $\psi'(\alpha_i) = \varphi(\psi(\alpha_i))$. We have,

$$\begin{array}{rclcll} \psi'(\varphi(\alpha_i)) & = & \psi'(\alpha_i) & = & \varphi(\psi(\alpha_i)) & \forall i, \text{ since } \alpha_i \text{ are out of reach of } \varphi \\ \psi'(\varphi(\beta)) & = & \varphi(\beta) & = & \varphi(\psi(\beta)) & \forall \beta \neq \alpha_i \end{array}$$

Thus $\psi'(\varphi(\tau_0)) = \varphi(\psi(\tau_0)) = \varphi(\tau)$, which implies that $\varphi(E(x)) \geq \varphi(\tau)$. This allows us to conclude $\varphi(E) \vdash x : \varphi(\tau)$ as desired.

Case 4: a is $(f \texttt{ where } f(x) = a)$, $(a_1, \ldots, a_n)$, or $(a_1\ a_2)$ — All these cases follow immediately using the induction hypothesis on their respective antecedents.

Case 5: a is $(\texttt{let } x = a_1 \texttt{ in } a_2)$ — The typing derivation ends in the LET rule:

$$\frac{E \vdash a_1 : \tau_1 \qquad E + \{x \mapsto \mathit{Gen}(E, \tau_1)\} \vdash a_2 : \tau_2}{E \vdash \texttt{let } x = a_1 \texttt{ in } a_2 : \tau_2}$$

By definition of generalization we have,

$$Gen(E, \tau_1) = \forall \alpha_1 \ldots \alpha_n .\, \tau_1 \quad \text{and} \quad \{\alpha_1 \ldots \alpha_n\} = \mathcal{F}(\tau_1) \setminus \mathcal{D}(\tau_1) \setminus \mathcal{F}(E) \tag{3.19}$$

Let $\beta_1 \ldots \beta_n$ be new variables that are out of reach of φ and are not free in E. Define a new substitution $\varphi' = \varphi \circ \{\alpha_i \mapsto \beta_i\}$. Using induction hypothesis we have,

$$\varphi'(E) \;\vdash\; a_1 : \varphi'(\tau_1) \tag{3.20}$$

$$\varphi(E) + \{x \mapsto \varphi(Gen(E, \tau_1))\} \;\vdash\; a_2 : \varphi(\tau_2) \tag{3.21}$$

Since no α_i is free in E, we have $\varphi'(E) = \varphi(E)$. Therefore, in order to apply the LET rule to the induction judgments 3.20 and 3.21 we need to show the following:

$$\varphi(Gen(E, \tau_1)) = Gen(\varphi'(E), \varphi'(\tau_1)) \tag{3.22}$$

We show this in two steps. Define $V = \mathcal{F}(\varphi'(\tau_1)) \setminus \mathcal{D}(\varphi'(\tau_1)) \setminus \mathcal{F}(\varphi'(E))$.

SubCase 5.1: $\{\beta_1 \ldots \beta_n\} \subseteq V$ — We follow the definition of V given above. We have,

1. $\beta_i \in \mathcal{F}(\varphi'(\tau_1))$ — From Proposition 3.8 we obtain $\mathcal{F}(\varphi'(\tau_1)) = \mathcal{F}(\varphi'(\mathcal{F}(\tau_1)))$, and for $\alpha_i \in \mathcal{F}(\tau_1)$ we have $\mathcal{F}(\varphi'(\alpha_i)) = \mathcal{F}(\beta_i) = \beta_i$.
2. $\beta_i \notin \mathcal{D}(\varphi'(\tau_1))$ — From Proposition 3.8 we obtain $\mathcal{D}(\varphi'(\tau_1)) \subseteq \mathcal{F}(\varphi'(\mathcal{D}(\tau_1))) \cup \mathcal{D}(\varphi'(\mathcal{F}(\tau_1)))$. Now we have,
   - $\beta_i \notin \mathcal{F}(\varphi'(\mathcal{D}(\tau_1)))$ — From Equation 3.19 $\alpha_i \notin \mathcal{D}(\tau_1)$ and for all $\alpha \neq \alpha_i$, $\beta_i \notin \mathcal{F}(\varphi'(\alpha))$ since β_i are chosen to be out of reach of φ.
   - $\beta_i \notin \mathcal{D}(\varphi'(\mathcal{F}(\tau_1)))$ — By definition $\mathcal{D}(\varphi'(\alpha_i)) = \mathcal{D}(\beta_i) = \phi$ and for all $\alpha \neq \alpha_i$, $\beta_i \notin \mathcal{D}(\varphi'(\alpha))$.
3. $\beta_i \notin \mathcal{F}(\varphi'(E))$ — From Proposition 3.8 we obtain $\mathcal{F}(\varphi'(E)) = \mathcal{F}(\varphi'(\mathcal{F}(E)))$. Now from Equation 3.19 $\alpha_i \notin \mathcal{F}(E)$ and for all $\alpha \neq \alpha_i$, $\beta_i \notin \mathcal{F}(\varphi'(\alpha))$.

SubCase 5.2: $V \subseteq \{\beta_1 \ldots \beta_n\}$ — Suppose we have a $\beta \in \mathcal{F}(\varphi'(\tau_1))$ such that $\beta \neq \beta_i$. We wish to show that $\beta \notin V$.

From Proposition 3.8 we obtain $\beta \in \mathcal{F}(\varphi'(\mathcal{F}(\tau_1)))$. Let $\alpha \in \mathcal{F}(\tau_1)$ be such that $\beta \in \mathcal{F}(\varphi'(\alpha))$. Now $\alpha \neq \alpha_i$, otherwise $\beta = \mathcal{F}(\varphi'(\alpha_i)) = \mathcal{F}(\beta_i) = \beta_i$. Using Equation 3.19 we must have one of the following situations:

1. $\alpha \in \mathcal{D}(\tau_1)$ — This implies that $\beta \in \mathcal{F}(\varphi'(\mathcal{D}(\tau_1))) \Longrightarrow \beta \in \mathcal{D}(\varphi'(\tau_1))$ using Proposition 3.8. It follows from the definition of V that $\beta \notin V$.
2. $\alpha \in \mathcal{F}(E)$ — This implies that $\beta \in \mathcal{F}(\varphi'(\mathcal{F}(E))) \Longrightarrow \beta \in \mathcal{F}(\varphi'(E))$ using Proposition 3.8. Again, it follows from the definition of V that $\beta \notin V$.

Combining the above two cases we obtain $V = \{\beta_1 \ldots \beta_n\}$. Now we have,

$$\begin{aligned} Gen(\varphi'(E), \varphi'(\tau_1)) &= \forall \beta_1 \ldots \beta_n .\, \varphi'(\tau_1) && \text{by definition of generalization} \\ &= \varphi(\forall \alpha_1 \ldots \alpha_n .\, \tau_1) && \text{by substitution over type schemes} \\ &= \varphi(Gen(E, \tau_1)) \end{aligned}$$

This is the desired result of Equation 3.22, so the LET rule can now be applied on the induction hypotheses 3.20 and 3.21.

Case 6: a is `(close` a`)` — The typing derivation ends in the CLOSE rule:

$$\frac{E \vdash a : \tau\ ref(r) \qquad r \notin (\mathcal{F}(E) \cup \mathcal{F}(\tau))}{E \vdash \texttt{close}\ a : \tau\ ref(\epsilon)}$$

Just as in the last case, let r' be a new region variable out of reach of φ and not free in E or τ. Define a new substitution $\varphi' = \varphi \circ \{r \mapsto r'\}$. Now we have,

$$\begin{array}{lrcll} & \varphi'(E) & \vdash & a : \varphi'(\tau\ \mathit{ref}(r)) & \text{by induction hypothesis} \\ \Longrightarrow & \varphi(E) & \vdash & a : (\varphi(\tau))\ \mathit{ref}(r') & \text{since } r \notin (\mathcal{F}(E) \cup \mathcal{F}(\tau)) \end{array} \quad (3.23)$$

It is also clear that $r' \notin (\mathcal{F}(\varphi'(E)) \cup \mathcal{F}(\varphi'(\tau)))$ since $r \notin (\mathcal{F}(E) \cup \mathcal{F}(\tau))$ and r' was chosen to be out of reach of φ. Therefore, we can apply the CLOSE rule to the induction hypothesis 3.23 to obtain the desired result.

□

The following proposition states that a typing remains valid under a more general typing environment.

Proposition 3.10 *Let a be an expression, τ be a type, and E, E' be two typing environments such that $Dom(E) = Dom(E')$, and $E'(x) \geq E(x)$ for all x free in a. If $E \vdash a : \tau$, then $E' \vdash a : \tau$.*

Proof: by simple structural induction over a. The base case for the IDENT rule follows directly from hypothesis, since $E'(x) \geq E(x) \geq \tau$. That is, any instance of $E(x)$ is also an instance of $E'(x)$. For the LET and CLOSE rules, we observe that $\mathcal{F}(E') \subseteq \mathcal{F}(E)$. In the LET rule, this implies that $Gen(E', \tau_1) \geq Gen(E, \tau_1)$ and the result follows by applying the induction hypothesis to the second antecedent. For the CLOSE rule, this implies that $r \notin \mathcal{F}(E')$ and the result follows directly. □

3.3 Type Soundness

3.3.1 Semantic Model

In order to show the soundness of the typing judgments generated by the above type system with respect to its evaluation rules, first, we must precisely characterize a "consistent" semantic relationship between value-domain entities and their corresponding type-domain entities. Since values may contain reference locations from the store, we need to define STORE TYPINGS which are finite mappings from store locations to types:

$$\text{STORE TYPINGS:} \qquad S \quad ::= \quad \{l_1 \mapsto \tau_1, \ldots, l_n \mapsto \tau_n\}$$

Note that we do not allow type schemes in store typings. This clearly separates the modeling of type generalization which is handled entirely *via* type environments, from the modeling of closing a mutable object which is handled entirely *via* type conversion within the store typing. Two store typings may be related by extension:

Definition 3.11 *A store typing S'* **extends** *another store typing S if $Dom(S) \subseteq Dom(S')$ and $S(l) = S'(l)$ for all $l \in Dom(S)$.*

Now, we define the following consistency relationships between value-domain entities and type-domain entities:

Definition 3.12 (Semantic Model) *Let s be a store, S be a store typing, e be an environment, E be a type environment, v be a value, τ be a type, and σ be a type scheme. Define the following relations:*

Case *1:* $S \models v : \tau$ — *The value* v *belongs to the type* τ *under the store typing* S. *The various cases are as follows:*

SubCase *1.1:* $S \models c : \mathit{typeof}(c)$, *where typeof is a predefined relation between predefined constants and their types.*

SubCase *1.2:* $S \models \langle \textsf{n-tup}\ v_1, \ldots, v_n \rangle : (\tau_1, \ldots, \tau_n)$, *if for all* i, $S \models v_i : \tau_i$.

SubCase *1.3:* $S \models \langle \textsf{clsr}\ f, x, a, e \rangle : \tau_1 \multimap\!\langle\pi\rangle\!\!\rightarrow \tau_2$, *if there exists a type environment* E *such that* $S \models e : E$ *and* $E \vdash (f\ \texttt{where}\ f(x) = a) : \tau_1 -\!\langle\pi\rangle\!\!\rightarrow \tau_2$.

SubCase *1.4:* $S \models l : \tau\ \mathit{ref}(r)$, *if* $l \in \mathit{Dom}(S)$ *and* $S(l) = \tau\ \mathit{ref}(r)$.

SubCase *1.5:* $S \models l : \tau\ \mathit{ref}(\epsilon)$, *if* $l \in \mathit{Dom}(S)$ *and there exists a substitution* φ *with* $\mathit{Dom}(\varphi) \subseteq \mathcal{F}(S(l)) \setminus \mathcal{D}(S(l))$ *such that* $\varphi(S(l)) = \tau\ \mathit{ref}(\epsilon)$.

Case *2:* $S \models v : \sigma$ — *The value* v *belongs to the type scheme* $\sigma = \forall \alpha_1 \ldots \alpha_n.\tau$ *under the store typing* S, *if none of* α_i *belong to* $\mathcal{D}(\tau)$ *and if* $S \models v : \varphi(\tau)$ *for all substitutions* φ *with* $\mathit{Dom}(\varphi) \subseteq \{\alpha_1 \ldots \alpha_n\}$.

Case *3:* $S \models e : E$ — *The values contained in the environment* e *belong to the corresponding type schemes in the type environment* E *(pointwise) under the store typing* S, *if* $\mathit{Dom}(e) = \mathit{Dom}(E)$ *and for all* $x \in \mathit{Dom}(E)$ *we have* $S \models e(x) : E(x)$.

Case *4:* $\models s : S$ — *The values contained in the store* s *belong to the corresponding types in the store typing* S *(pointwise), if* $\mathit{Dom}(s) = \mathit{Dom}(S)$ *and for all* $l \in \mathit{Dom}(S)$ *we have,*

SubCase *4.1: If* $S(l) = \tau\ \mathit{ref}(r)$ *then* $s(l) = v, \mathrm{rw}$ *and* $S \models v : \tau$.

SubCase *4.2: If* $S(l) = \tau\ \mathit{ref}(\epsilon)$ *then* $s(l) = v, \mathrm{ro}$ *and* $S \models v : \tau$.

The primary motivation of "closing" a mutable object is to be able to generalize its type to a type scheme and use it like any other functional value in a safe manner. This is modeled in Case 1.5 by defining a closed location to be consistent with any type obtained *via* a substitution over the non-dangerous variables of the type present in the store typing. On the other hand in Case 1.4, a mutable location is defined to be consistent only with the exact type present in the store typing, modeling the fact that it is allowed to have only a monomorphic type. The one-to-one correspondence between dynamic mutability of a reference location and its type is reflected in Cases 4.1 and 4.2. Only the locations with a read/write tag are defined to be consistent with a mutable reference type and vice-versa.

3.3.2 Properties of the Semantic Model

During the course of evaluation of a program, the values contained within the store locations may change but the types of those locations remain the same (except for the types of locations that are currently being closed). This fact is useful in showing that a semantic relation such as $S \models v : \tau$ that holds true at some point during evaluation, remains true afterwards under any extension of the current store typing:

Proposition 3.13 (Store Typing Extension) *If* S' *extends* S, *then* $S \models v : \tau$ *implies* $S' \models v : \tau$. *Similarly,* $S \models e : E$ *implies* $S' \models e : E$.

Proof: by a simple induction over v. The only interesting case is that for locations. The definition of extension ensures that S and S' must agree exactly on the types of the locations that are present in S. □

3.3.3 Type Soundness

Before we establish the consistency of the static and the dynamic semantics in terms of a soundness theorem, it is useful to characterize the semantic meaning of the generalization and the closing operations in terms of the above semantic definitions.

The following proposition establishes the fact that it is semantically safe to generalize the non-dangerous variables of a type.

Proposition 3.14 (Semantic Generalization) *Let v be a value, τ be a type and S be a store typing such that $S \models v : \tau$. Let $\alpha_1, \ldots, \alpha_m$ be type variables such that for all i, $\alpha_i \notin \mathcal{D}(\tau)$. Then, for all substitutions φ with $Dom(\varphi) \subseteq \{\alpha_1 \ldots \alpha_m\}$, we have $S \models v : \varphi(\tau)$. As a consequence, $S \models v : \forall \alpha_1 \ldots \alpha_m . \tau$.*

Proof: by structural induction over v. Only the case for a closed location is different from [Ler92], but we show all cases for the sake of completeness.

Case 1: v is c — By definition, $S \models c : \mathit{typeof}(c)$ and therefore we must have $\mathit{typeof}(c) \geq \tau$ using the hypothesis $S \models c : \tau$. Also by assumption, all predefined constants possess fully quantified type schemes, *i.e.*, their type schemes do not contain any free type variables. This implies $\mathit{typeof}(c) \geq \varphi(\tau)$ and the result $S \models c : \varphi(\tau)$ follows immediately.

Case 2: v is $\langle \textsf{n-tup}\ v_1, \ldots, v_n \rangle$ and τ is $\tau_1, \ldots, \tau_n$ — Since $\mathcal{D}(\tau_1, \ldots, \tau_n) = \bigcup_{1 \leq j \leq n} \mathcal{D}(\tau_j)$, we must have for all i, j that $\alpha_i \notin \tau_j$. By induction hypothesis it follows that for all j, $S \models v_j : \varphi(\tau_j)$. The result follows from the definition of $\models$ for tuples.

Case 3: v is $\langle \textsf{clsr}\ f, x, a, e \rangle$ and τ is $\tau_1 \multimap\!\langle\pi\rangle\!\rightarrowtail \tau_2$ — Applying the definition of $\models$ for closures, let E be the type environment such that,

$$S \models e : E \tag{3.24}$$
$$E \vdash (f\ \texttt{where}\ f(x) = a) : \tau_1 \multimap\!\langle\pi\rangle\!\rightarrowtail \tau_2 \tag{3.25}$$

We will show that,

$$S \models e : \varphi(E) \tag{3.26}$$
$$\varphi(E) \vdash (f\ \texttt{where}\ f(x) = a) : \varphi(\tau_1 \multimap\!\langle\pi\rangle\!\rightarrowtail \tau_2) \tag{3.27}$$

Equation 3.27 follows directly from Equation 3.25 using Proposition 3.9 that typing is stable under substitution. Also note that $Dom(E) = Dom(e) = \mathcal{F}(f\ \texttt{where}\ f(x) = a)$ from Equation 3.24 and the dynamic ABS rule in Figure 3.1.

In order to show Equation 3.26, we must show $S \models e(y) : \varphi(E(y))$ for all $y \in Dom(E)$. For a given y, let $E(y) = \forall \beta_1 \ldots \beta_k.\ \tau'$ where β_j are taken out of reach of φ and distinct from α_i. Using substitution over type schemes, we obtain $\varphi(E(y)) = \forall \beta_1 \ldots \beta_k.\ \varphi(\tau')$. Thus, in order to conclude $S \models e(y) : \varphi(E(y))$, first we have to show $S \models e(y) : \psi(\varphi(\tau'))$ for any substitution ψ with $Dom(\psi) \subseteq \{\beta_1 \ldots \beta_k\}$. This is done as follows.

From Equation 3.24 we obtain $S \models e(y) : E(y)$ which implies $S \models e(y) : \tau'$ using the definition of $\models$ over type schemes. Now consider the substitution $\psi \circ \varphi$. Its domain is $\{\alpha_1, \ldots, \alpha_n, \beta_1, \ldots, \beta_k\}$. We claim that none of these variables are dangerous in τ':

- $\alpha_i \notin \mathcal{D}(\tau')$ — We know that $y \in \mathcal{F}(f\ \texttt{where}\ f(x) = a)$, so its type scheme $E(y)$ is included in the closure type π of $\tau = \tau_1 \multimap\!\langle\pi\rangle\!\rightarrowtail \tau_2$. This implies that $\mathcal{D}(E(y)) = \mathcal{D}(\tau') \setminus \{\beta_1 \ldots \beta_k\}$ is included in $\mathcal{D}(\tau) = \mathcal{D}(\pi)$. Since $\alpha_i \notin \mathcal{D}(\tau)$ by hypothesis, it follows that $\alpha_i \notin \mathcal{D}(\tau')$ for all i.

- $\beta_j \notin \mathcal{D}(\tau')$ — $S \models e(y) : E(y)$ from Equation 3.24 immediately implies $\beta_j \notin \mathcal{D}(\tau')$ for all j.

Now we can apply the induction hypothesis to the value $e(y)$, the type τ', the variables $\alpha_1, \ldots, \alpha_n, \beta_1, \ldots, \beta_k$ and the substitution $\psi \circ \varphi$ to obtain $S \models e(y) : \psi(\varphi(\tau'))$. This holds for any substitution ψ over $\{\beta_1 \ldots \beta_k\}$. Moreover, none of β_j are dangerous in $\varphi(\tau')$ since they are not dangerous in τ' and they are out of reach of φ. Therefore we obtain $S \models e(y) : \forall \beta_1 \ldots \beta_k.\ \tau'$ by definition of $\models$ over type schemes, that is, $S \models e(y) : \varphi(E(y))$. This holds for all $y \in Dom(E)$. Hence Equation 3.26 is satisfied and we obtain the desired result.

Case 4: v is l and τ is $\tau_1\ ref(r)$ — Here, $\mathcal{D}(\tau) = \mathcal{F}(\tau)$. Since no α_i is dangerous in τ by hypothesis, it follows that no α_i can be free in τ. Thus, $\varphi(\tau) = \tau$ and the result follows immediately from the hypothesis that $S \models v : \tau$.

Case 5: v is l and τ is $\tau_1\ ref(\epsilon)$ — Applying the definition of $\models$ for non-mutable locations, let ψ be the substitution with $Dom(\psi) \subseteq \mathcal{F}(S(l)) \setminus \mathcal{D}(S(l))$ such that $\tau = \psi(S(l))$ thereby implying $\varphi(\tau) = \varphi(\psi(S(l)))$. Also, no $\alpha_i \in Dom(\varphi)$ is dangerous in $S(l)$. Otherwise, it would surely be dangerous in $\tau = \psi(S(l))$ from Proposition 3.8 which contradicts the hypothesis.

Consider the substitution $\varphi \circ \psi$ restricted to the domain $X = \mathcal{F}(S(l)) \setminus \mathcal{D}(S(l))$. From the above remarks it is clear that we still have $(\varphi \circ \psi) \mid_X (S(l)) = \varphi(\tau)$. Thus, we can apply the definition of $\models$ for the location l using the substitution $(\varphi \circ \psi) \mid_X$ to conclude $S \models l : \varphi(\tau)$ as desired.

□

The following proposition establishes a correspondence between the dangerous regions of a type and the mutable locations that are reachable from a value possessing that type. This allows us to use dangerous regions as a safe static abstraction for mutable locations.

Proposition 3.15 (Region Abstraction) *Let s be a store, and S be a store typing such that $\models s : S$. Then we have,*

$$S \models v : \tau \implies \left(\bigcup_{l \in Reachable(v,s)} \mathcal{R}(S(l)) \right) \subseteq \mathcal{R}(\tau)$$

That is, the dangerous regions contained in the types of reachable locations of a value are dangerous in the type of the value. Using pointwise extension to environments we also have,

$$S \models e : E \implies \left(\bigcup_{l \in Reachable(e,s)} \mathcal{R}(S(l)) \right) \subseteq \mathcal{R}(E)$$

Proof: by induction on the *depth* of reachability of a location l in the value v. First, we define a family of reachability functions $Reachable^i(v, s)$ as follows:

$$Reachable^0(v, s) \quad = \quad \mathcal{L}(v) \tag{3.28}$$

$$Reachable^{i+1}(v, s) \quad = \quad Reachable^i(v, s) \bigcup \left(\bigcup_{l \in Reachable^i(v,s)} \mathcal{L}(value(s(l))) \right) \tag{3.29}$$

By definition, $Reachable(v,s)$ is the limit of the increasing chain of sets $Reachable^0(v,s) \subseteq Reachable^1(v,s) \subseteq \cdots$. Since the number of locations reachable from a value is finite, this chain is guaranteed to reach the limit at a finite i. We will show that for all i,

$$\left(\bigcup_{l \in Reachable^i(v,s)} \mathcal{R}(S(l)) \right) \subseteq \mathcal{R}(\tau) \tag{3.30}$$

Base Case: Using Equation 3.28, we need to show that for all locations $l \in \mathcal{L}(v)$, we have $\mathcal{R}(S(l)) \subseteq \mathcal{R}(\tau)$. This is shown by induction on the structure of v using the definition of $S \models v : \tau$.

Case 1: v is c — Trivial, since there are no locations reachable from a constant.

Case 2: v is $\langle \textsf{n-tup}\ v_1, \ldots, v_n \rangle$ and τ is $\tau_1, \ldots, \tau_n$ — Follows immediately from the definition of $\models$ for tuples and the induction hypothesis for each v_i.

Case 3: v is $\langle \textsf{clsr}\ f, x, a, e \rangle$ and τ is $\tau_1 \prec\!\langle \pi \rangle\!\!\rightarrowtail \tau_2$ — From the definition of $\models$ for closures we obtain that there exists a type environment E such that,

$$S \models e : E \quad \text{and} \quad E \vdash (f\ \texttt{where}\ f(x) = a) : \tau_1 \prec\!\langle \pi \rangle\!\!\rightarrowtail \tau_2 \tag{3.31}$$

Applying the definition of reachability for $\langle \textsf{clsr}\ f, x, a, e \rangle$ and the induction hypothesis for environments we obtain:

$$\left(\bigcup_{l \in Reachable(v,s)} \mathcal{R}(S(l)) \right) = \left(\bigcup_{l \in Reachable(e,s)} \mathcal{R}(S(l)) \right) \subseteq \mathcal{R}(E) \tag{3.32}$$

The desired result follows by noticing that $\mathcal{R}(E) \subseteq \mathcal{R}(\tau)$ since $Dom(E) = Dom(e) = \mathcal{F}(f\ \texttt{where}\ f(x) = a)$ and all the type schemes in $CoDom(E)$ are included in the closure type of τ by construction.

Case 4: v is l and τ is $\tau_1\ ref(r)$ — Follows immediately from the definition of $\models$ for mutable locations since $S(l) = \tau$.

Case 5: v is l and τ is $\tau_1\ ref(\epsilon)$ — From the definition of $\models$ for non-mutable locations we have $\varphi(S(l)) = \tau$. But, the domain of φ does not include any dangerous variables of $S(l)$, so we must have $\mathcal{R}(S(l)) \subseteq \mathcal{R}(\varphi(S(l))) = \mathcal{R}(\tau)$ as desired.

Induction Case: We assume the hypothesis for i,

$$\left(\bigcup_{l \in Reachable^i(v,s)} \mathcal{R}(S(l)) \right) \subseteq \mathcal{R}(\tau) \tag{3.33}$$

From Equation 3.29, the locations in $Reachable^i(v,s)$ are already covered *via* the above hypothesis. Given a location $l \in Reachable^i(v,s)$, let $value(s(l)) = v'$ and $S(l) = \tau'\ ref(\rho)$. Using hypothesis $\models s : S$, we have $S \models v' : \tau'$. Therefore, we can apply the base case induction as above and obtain for all $l' \in \mathcal{L}(v')$, $\mathcal{R}(S(l')) \subseteq \mathcal{R}(\tau')$. This immediately extends to $\mathcal{R}(S(l')) \subseteq \mathcal{R}(\tau)$, since τ' is contained in $S(l)$ and $\mathcal{R}(S(l)) \subseteq \mathcal{R}(\tau)$ from Equation 3.33.

□

The semantic consistency between the static and the dynamic semantics can now be stated in the form of the soundness theorem given below. It is proved using induction on the size of evaluation derivation, doing a case analysis on a and hence on the last rule used in the typing derivation.

The soundness of the `close` operation relies on the fact that it only closes *fresh* and *non-escaping* locations, *i.e.*, locations that are neither present in the initial store, nor are accessible from the current environment or the returned result. The former is a property of the dynamic rules (Proposition 3.5) and the latter is ensured by the side condition on the static CLOSE rule and Proposition 3.15.

Theorem 3.16 (Type Soundness) *Let a be an expression, τ be a type, E be a type environment, e be an evaluation environment, s be an initial store, and S be a store typing such that:*

$$E \vdash a : \tau \qquad \textit{and} \qquad S \models e : E \qquad \textit{and} \qquad \models s : S$$

If there exists a result r such that $e \vdash a/s \Rightarrow r$, then $r \neq \texttt{err}$, instead $r = v/s'$ for some value v and a resulting store s', and there exists a store typing S' such that:

$$S' \textit{ extends } S \qquad \textit{and} \qquad S' \models v : \tau \qquad \textit{and} \qquad \models s' : S'$$

Proof: by induction on the size of evaluation derivation. We argue by case analysis on a and hence on the last rule used in the typing derivation. Again, only the case for the CLOSE rule is different from [Ler92], but we show all cases.

Case 1: Constants — The typing rule is:

$$\frac{\textit{typeof}(c) \geq \tau}{E \vdash c : \tau}$$

The only possible evaluation is $e \vdash c/s \Rightarrow c/s$. By definition of $\models$ for constants, we have $S \models c : \textit{typeof}(c)$ which implies $S \models c : \tau$ since $\textit{typeof}(c) \geq \tau$. We conclude with $S' = S$.

Case 2: Variables — The typing rule is:

$$\frac{x \in \textit{Dom}(E) \qquad E(x) \geq \tau}{E \vdash x : \tau}$$

From hypothesis $S \models e : E$ it follows that $x \in \textit{Dom}(e)$ and $S \models e(x) : E(x)$. Thus, the only possible evaluation is $e \vdash x/s \Rightarrow e(x)/s$. By definition of $\models$ for type schemes, $S \models e(x) : E(x)$ implies $S \models e(x) : \tau$. We conclude with $S' = S$.

Case 3: Function Abstraction — The typing rule is:

$$\frac{\{y_1 \ldots y_n\} = \mathcal{F}(f \texttt{ where } f(x) = a) \qquad E + \{f \mapsto \tau_1 \multimap\!\langle E(y_1), \ldots, E(y_n), \pi\rangle\!\!\rightarrow \tau_2, x \mapsto \tau_1\} \vdash a : \tau_2}{E \vdash (f \texttt{ where } f(x) = a) : \tau_1 \multimap\!\langle E(y_1), \ldots, E(y_n), \pi\rangle\!\!\rightarrow \tau_2}$$

The only possible evaluation is $e \vdash (f \texttt{ where } f(x) = a)/s \Rightarrow \langle \mathsf{clsr}\ f, x, a, e\,|_Y\rangle/s$ where $Y = \{y_1 \ldots y_n\}$. Using the definition of $\models$ for closures, we have $S \models \langle \mathsf{clsr}\ f, x, a, e\,|_Y\rangle : \tau_1 \multimap\!\langle\pi\rangle\!\!\rightarrow \tau_2$ taking $E\,|_Y$ to be the desired type environment. We conclude with $S' = S$.

Case 4: Function Application — The typing rule is:

$$\frac{E \vdash a_1 : \tau_1 \multimap\!\langle\pi\rangle\!\!\rightarrow \tau_2 \qquad E \vdash a_2 : \tau_1}{E \vdash a_1\ a_2 : \tau_2}$$

We claim that evaluations leading to `err` are not possible and that the following evaluation rule applies:

$$\frac{e \vdash a_1/s \Rightarrow \langle \mathsf{clsr}\ f, x, a_0, e_0\rangle/s_1 \qquad e \vdash a_2/s_1 \Rightarrow v_2/s_2 \qquad e_0 + \{f \mapsto \langle \mathsf{clsr}\ f, x, a_0, e_0\rangle, x \mapsto v_2\} \vdash a_0/s_2 \Rightarrow v/s'}{e \vdash (a_1\ a_2)/s \Rightarrow v/s'}$$

This is shown as follows:

Using the induction hypothesis on a_1, we obtain that it cannot evaluate to `err`, instead it must evaluate to a closure, $e \vdash a_1/s \Rightarrow \langle \mathsf{clsr}\ f, x, a_0, e_0\rangle/s_1$ with a store typing S_1 such that:

$$S_1 \models \langle \mathsf{clsr}\ f, x, a_0, e_0\rangle : \tau_1 \stackrel{\langle\pi\rangle}{\longrightarrow} \tau_2 \quad \text{and} \quad \models s_1 : S_1 \quad \text{and} \quad S_1 \text{ extends } S \tag{3.34}$$

Since S_1 extends S, we have $S_1 \models e : E$ from Proposition 3.13. Thus, we can use the induction hypothesis on a_2 with store $s_1 : S_1$ and obtain that it evaluates to a proper value as well, $e \vdash a_2/s_1 \Rightarrow v_2/s_2$ with a store typing S_2 such that:

$$S_2 \models v_2 : \tau_1 \qquad \text{and} \qquad \models s_2 : S_2 \qquad \text{and} \qquad S_2 \text{ extends } S_1 \tag{3.35}$$

Applying the definition of $\models$ to the first clause in Equation 3.34, we obtain that there exists a type environment E_0 such that:

$$\begin{array}{lrcl} & S_1 & \models & e_0 : E_0 \qquad (3.36) \\ \text{and} & E_0 & \vdash & (f\ \mathtt{where}\ f(x) = a_0) : \tau_1 \stackrel{\langle\pi\rangle}{\longrightarrow} \tau_2 \\ \Longrightarrow & E_0 + \{f \mapsto \tau_1 \stackrel{\langle\pi\rangle}{\longrightarrow} \tau_2, x \mapsto \tau_1\} & \vdash & a_0 : \tau_2 \qquad (3.37) \end{array}$$

Now consider the following environments:

$$e_2 = e_0 + \{f \mapsto \langle \mathsf{clsr}\ f, x, a_0, e_0\rangle, x \mapsto v_2\} \qquad \text{and} \qquad E_2 = E_0 + \{f \mapsto \tau_1 \stackrel{\langle\pi\rangle}{\longrightarrow} \tau_2, x \mapsto \tau_1\}$$

Using Proposition 3.13 and Equations 3.34, 3.35, and 3.36, we obtain $S_2 \models e_2 : E_2$. Therefore, we can apply the induction hypothesis to the typing judgment 3.37 and the store $s_2 : S_2$. We obtain the evaluation $e_2 \vdash a_0/s_2 \Rightarrow v/s'$ with a store typing S' such that:

$$S' \models v : \tau_2 \qquad \text{and} \qquad \models s' : S' \qquad \text{and} \qquad S' \text{ extends } S_2 \tag{3.38}$$

This shows that a_0 in the third premise of the evaluation rule given above also evaluates to a proper value and we obtain the desired result since S' extends S by transitivity.

Case 5: **Tuple Construction** — Same argument as above.

Case 6: **`let`-binding** — The typing rule is:

$$\frac{E \vdash a_1 : \tau_1 \qquad E + \{x \mapsto \mathit{Gen}(E, \tau_1)\} \vdash a_2 : \tau_2}{E \vdash \mathtt{let}\ x = a_1\ \mathtt{in}\ a_2 : \tau_2}$$

Again, we claim that evaluations leading to `err` are not possible and the last step in evaluation derivation is:

$$\frac{e \vdash a_1/s \Rightarrow v_1/s_1 \qquad e + \{x \mapsto v_1\} \vdash a_2/s_1 \Rightarrow v_2/s'}{e \vdash (\mathtt{let}\ x = a_1\ \mathtt{in}\ a_2)/s \Rightarrow v_2/s'}$$

This is shown as follows:

Using the induction hypothesis on a_1, we obtain that it does not evaluate to `err`, instead $e \vdash a_1/s \Rightarrow v_1/s_1$ with the store typing S_1 such that:

$$S_1 \models v_1 : \tau_1 \qquad \text{and} \qquad \models s_1 : S_1 \qquad \text{and} \qquad S_1 \text{ extends } S \tag{3.39}$$

Using Proposition 3.14, we have $S_1 \models v_1 : \mathit{Gen}(E, \tau_1)$ since the Gen operator does not generalize any dangerous variables in τ_1. Now, consider the following environments:

$$e_1 = e + \{x \mapsto v_1\} \qquad \text{and} \qquad E_1 = E + \{x \mapsto \mathit{Gen}(E, \tau_1)\}$$

Since S_1 extends S, we obtain $S_1 \models e_1 : E_1$. Therefore, we can apply the induction hypothesis to the second premise of the typing rule and the store $s_1 : S_1$ to obtain $e_1 \vdash a_2/s_1 \Rightarrow v_2/s'$ with the store typing S' such that:

$$S' \models v_2 : \tau_2 \qquad \text{and} \qquad \models s' : S' \qquad \text{and} \qquad S' \text{ extends } S_1 \tag{3.40}$$

This is the desired result.

Case 7: **Reference Creation** — The PRIMAPP typing rule instantiates to:

$$\frac{\forall t, u, r.\, t \multimap\!\langle u \rangle\!\mapsto t \; \mathit{ref}(r) \geq \tau \multimap\!\langle \pi \rangle\!\mapsto \tau \; \mathit{ref}(r) \qquad E \vdash a : \tau}{E \vdash \mathtt{ref}(a) : \tau \; \mathit{ref}(r)}$$

The evaluation must end up with:

$$\frac{e \vdash a/s \Rightarrow v/s_1 \qquad l \notin \mathit{Dom}(s_1)}{e \vdash \mathtt{ref}(a)/s \Rightarrow l/(s_1 + \{l \mapsto v, \mathrm{rw}\})}$$

By induction hypothesis applied to a, we obtain a store typing S_1 such that:

$$S_1 \models v : \tau \qquad \text{and} \qquad \models s_1 : S_1 \qquad \text{and} \qquad S_1 \text{ extends } S \tag{3.41}$$

Let us define,

$$s' = s_1 + \{l \mapsto v, \mathrm{rw}\} \qquad \text{and} \qquad S' = S_1 + \{l \mapsto \tau \; \mathit{ref}(r)\}$$

Since $\mathit{Dom}(s_1) = \mathit{Dom}(S_1)$, we have $l \notin \mathit{Dom}(S_1)$. Hence, S' extends S_1 and therefore S. Using this fact on the first clause of Equation 3.41, we obtain $S' \models v : \tau$, which allows us to conclude from the definition of $\models$ that $S' \models l : (\tau \; \mathit{ref}(r))$ and $\models s' : S'$.

Case 8: **Dereferencing** — We show the case for dereferencing a non-mutable location. The case of dereferencing a mutable location is similar. The PRIMAPP typing rule instantiates to:

$$\frac{\forall t, u.\, t \; \mathit{ref}(\epsilon) \multimap\!\langle u \rangle\!\mapsto t \geq \tau \; \mathit{ref}(\epsilon) \multimap\!\langle \pi \rangle\!\mapsto \tau \qquad E \vdash a : \tau \; \mathit{ref}(\epsilon)}{E \vdash \mathtt{!}a : \tau}$$

By induction hypothesis applied to a, we obtain that it must evaluate to a location $e \vdash a/s \Rightarrow l/s_1$ with a store typing S_1 such that:

$$S_1 \models l : \tau \; \mathit{ref}(\epsilon) \qquad \text{and} \qquad \models s_1 : S_1 \qquad \text{and} \qquad S_1 \text{ extends } S \tag{3.42}$$

Also, $l \in \mathit{Dom}(s_1)$ because the first clause above implies that $l \in \mathit{Dom}(S_1)$ and $\mathit{Dom}(s_1) = \mathit{Dom}(S_1)$ from the second clause. Thus, the only possibility for evaluation is:

$$\frac{e \vdash a/s \Rightarrow l/s_1 \qquad l \in \mathit{Dom}(s_1) \qquad \mathit{value}(s_1(l)) = v}{e \vdash \mathtt{!}a/s \Rightarrow v/s_1}$$

Applying the definition of $\models$ for non-mutable locations to the first clause in Equation 3.42, we obtain that there exists a substitution ψ with $\mathit{Dom}(\psi) \subseteq \mathcal{F}(S_1(l)) \setminus \mathcal{D}(S_1(l))$ such that $\psi(S_1(l)) = \tau \; \mathit{ref}(\epsilon)$. Thus, $S_1(l)$ must be of the form:

$$S_1(l) = \tau' \; \mathit{ref}(\epsilon) \qquad \text{with} \qquad \psi(\tau') = \tau$$

This is because all locations must have reference types and we never substitute the null region for region variables. From the definition of $\models s_1 : S_1$ for location l it follows that $S_1 \models v : \tau'$. Since $\mathit{Dom}(\psi)$ does not include any dangerous variables in $S(l)$ and hence in τ', we can apply Proposition 3.14 to substitution ψ and obtain $S_1 \models v : \psi(\tau')$. This is the desired result taking $S' = S_1$.

Case 9: **Assignment** — The PRIMAPP typing rule instantiates to:

$$\frac{\forall t, u, r.\ (t\ \mathit{ref}(r), t) \multimap\!\langle u\rangle\!\!\rightarrow \mathit{unit} \geq (\tau\ \mathit{ref}(r), \tau) \multimap\!\langle \pi\rangle\!\!\rightarrow \mathit{unit} \qquad E \vdash a : \tau\ \mathit{ref}(r), \tau}{E \vdash \texttt{:=}(a) : \mathit{unit}}$$

As in the previous case, the evaluation must end up with:

$$\frac{e \vdash a/s \Rightarrow (l, v)/s_1 \qquad l \in \mathit{Dom}(s_1) \qquad \mathit{tag}(s_1(l)) = \mathrm{rw}}{e \vdash \texttt{:=}(a)/s \Rightarrow \texttt{()}/(s_1 + \{l \mapsto v, \mathrm{rw}\})}$$

By induction hypothesis applied to a, we get a store typing S_1 such that:

$$S_1 \models (l, v) : \tau\ \mathit{ref}(r), \tau \qquad \text{and} \qquad \models s_1 : S_1 \qquad \text{and} \qquad S_1 \text{ extends } S \tag{3.43}$$

This implies $S_1 \models v : \tau$ and $S_1(l) = \tau\ \mathit{ref}(r)$ from the definition of $\models$ for tuples and mutable locations. Letting $S' = S_1$ and $s' = s_1 + \{l \mapsto v, \mathrm{rw}\}$, we therefore obtain $\models s' : S'$ using the definition of $\models$. Finally, we check that $S' \models \texttt{()} : \mathit{unit}$ and obtain the desired result.

Case 10: **`close` expression** — The typing rule is:

$$\frac{E \vdash a : \tau\ \mathit{ref}(r) \qquad r \notin (\mathcal{F}(E) \cup \mathcal{F}(\tau))}{E \vdash \texttt{close}\ a : \tau\ \mathit{ref}(\epsilon)}$$

Using the induction hypothesis on a, we obtain $e \vdash a/s \Rightarrow l/s_1$ with the store typing S_1 such that:

$$S_1 \models l : \tau\ \mathit{ref}(r) \qquad \text{and} \qquad \models s_1 : S_1 \qquad \text{and} \qquad S_1 \text{ extends } S \tag{3.44}$$

From the first two clauses above and the definition of $\models$ for mutable locations, we obtain,

$$l \in \mathit{Dom}(S_1) \qquad S_1(l) = \tau\ \mathit{ref}(r) \qquad s_1(l) = v, \mathrm{rw} \qquad \text{and} \qquad S_1 \models v : \tau \tag{3.45}$$

Thus, the CLOSE evaluation rule applies:

$$\frac{\begin{array}{c} e \vdash a/s \Rightarrow l/s_1 \qquad s_1(l) = v, \mathrm{rw} \\ L = \mathit{Reachable}(l, s_1) \cup \mathit{Reachable}(e, s_1) \cup \bigcup_{l' \in \mathit{Dom}(s)} \mathit{Reachable}(l', s_1) \end{array}}{e \vdash (\texttt{close}\ a)/s \Rightarrow l/(s_1 |_L + \{l \mapsto v, \mathrm{ro}\})}$$

Let us now define,

$$s' = s_1 |_L + \{l \mapsto v, \mathrm{ro}\} \qquad \text{and} \qquad S' = S_1 |_L + \{l \mapsto \tau\ \mathit{ref}(\epsilon)\} \tag{3.46}$$

Now, we have to show the following:

$$S' \models l : \tau\ \mathit{ref}(\epsilon) \qquad \text{and} \qquad \models s' : S' \qquad \text{and} \qquad S' \text{ extends } S \tag{3.47}$$

The first clause follows directly from the definition of $\models$ for non-mutable locations since we have chosen $l \in \mathit{Dom}(S')$ and $S'(l) = \tau\ \mathit{ref}(\epsilon)$.

Next, we show that S' extends S. Note that S_1 extends S from Equation 3.44 and $\mathit{Dom}(S) \subseteq \mathit{Dom}(S')$ by construction. Therefore, S' will extend S if $l \notin \mathit{Dom}(S)$, since that is the only location at which S_1 and S' differ. This is shown as follows.

Suppose for the moment that $l \in Dom(S)$. Since $\models s : S$ by hypothesis, we have $l \in Dom(s)$. Applying Proposition 3.5 to the evaluation $e \vdash a/s \Rightarrow l/s_1$ we conclude that $l \in Reachable(e, s)$. Also, since S_1 extends S, we obtain that $S(l) = S_1(l) = \tau\ ref(r)$. Finally, using Proposition 3.15 for the hypothesis $S \models e : E$, we must have $r \in \mathcal{R}(S(l)) \subseteq \mathcal{R}(E) \subseteq \mathcal{F}(E)$ which contradicts the condition $r \notin \mathcal{F}(E)$ in the typing rule.

As the final step in proving Equation 3.47, we have to show $\models s' : S'$. By construction, we have $Dom(s') = Dom(S')$ and at location l, $s'(l)$ has the read-only tag which is consistent with $S'(l)$ pointing to a null region. At locations $l' \in Dom(S')$ other than l, the tags in s' are already consistent with the corresponding regions in S' since they are directly copied from $s_1 : S_1$. Next, we have to show that for all locations $l' \in Dom(S')$ such that $value(s'(l')) = v'$ and $S'(l') = \tau'\ ref(\rho)$, we have,

$$S_1 \models v' : \tau' \Longrightarrow S' \models v' : \tau' \tag{3.48}$$

This can be shown by a simple structural induction on v'. Only the case for locations is interesting. By construction, the store s' is closed under reachability so there is no possibility of encountering undefined locations within v', and for locations other than l, we already have $S'(l') = S_1(l')$.

The only problem is if v' contained l (the location being closed), then τ' would still contain the region variable r because $S_1(l) = \tau\ ref(r)$. But this region has been closed in S', making $S' \models v' : \tau'$ inconsistent. Thus, l should not be contained in v'. This is where the domain restriction on the store s' proves useful. We show below a stronger condition that the location l is not *reachable* from any value v' present in the store s'. Specifically, we will show that $l \notin Reachable(v', s_1)$ which implies $l \notin Reachable(v', s')$.

Let us assume for the purpose of contradiction that $l \in Reachable(v', s_1)$. Looking at the components of $Dom(s')$ given by Equation 3.46, the following possibilities arise for $l' \in Dom(s')$:

1. $l' = l$ — Then $v' = v$ and hence $l \in Reachable(v, s_1)$ by assumption. Now, we apply Proposition 3.15 to $S_1 \models v : \tau$ taken from Equation 3.45 to conclude that $r \in \mathcal{R}(S_1(l)) \subseteq \mathcal{R}(\tau) \subseteq \mathcal{F}(\tau)$ which contradicts the condition $r \notin \mathcal{F}(\tau)$ in the typing rule.
2. $l' \neq l$ but $l' \in Reachable(l, s_1)$ — This immediately implies $l' \in Reachable(v, s_1)$ since the location l contains the value v in both s_1 and s'. Together with the assumption $l \in Reachable(v', s_1)$ and transitivity of reachability, we obtain $l \in Reachable(v, s_1)$ which leads to a contradiction as shown in the previous case.
3. $l' \in Reachable(e, s_1)$ — Using the assumption $l \in Reachable(v', s_1)$ and the fact that l' contains v' in s_1, we obtain by transitivity that $l \in Reachable(e, s_1)$. Applying Proposition 3.15 to $S_1 \models e : E$ derived from Equation 3.44 and Proposition 3.13, we conclude that $r \in \mathcal{R}(S_1(l)) \subseteq \mathcal{R}(E) \subseteq \mathcal{F}(E)$ which contradicts the condition $r \notin \mathcal{F}(E)$ in the typing rule.
4. $l' \in Reachable(Dom(s), s_1)$ — We know that l was not reachable from any value present in the domain s initially, *i.e.*, $l \notin Reachable(Dom(s), s)$ because we have already shown that $l \notin Dom(s)$ while showing that S' extends S. Thus, the only way l could become reachable from $Dom(s)$ after the evaluation $e \vdash a/s \Rightarrow v/s_1$ is if some location in $Dom(s)$ was assigned a new value from which l was reachable. Without loss of generality, let us assume that location is l' and the newly assigned value is v', *i.e.*,

$$\exists\, l' \in Dom(s) : \quad value(s(l')) \neq value(s_1(l')) = v' \quad \text{and} \quad l \in Reachable(v', s_1) \tag{3.49}$$

Since location l' was modified during the evaluation $e \vdash a/s \Rightarrow v/s_1$, we can apply Proposition 3.6 to conclude that $l' \in \mathit{Reachable}(e, s)$. Applying Proposition 3.15 to hypothesis $S \models e : E$, we obtain that $\mathcal{R}(S(l')) \subseteq \mathcal{R}(E)$ which extends to $\mathcal{R}(S_1(l')) \subseteq \mathcal{R}(E)$ since S_1 extends S.

On the other hand, from Equation 3.48 we already have $S_1 \models v' : \tau'$ where $S_1(l') = \tau'\ \mathit{ref}(r')$ for some region variable r'. Applying Proposition 3.15 in this case for the location $l \in \mathit{Reachable}(v', s_1)$ we obtain $r \in \mathcal{R}(S_1(l)) \subseteq \mathcal{R}(\tau') \subseteq \mathcal{R}(S_1(l'))$. Combining this with the result obtained in the last paragraph, we conclude $r \in \mathcal{R}(E) \subseteq \mathcal{F}(E)$ which contradicts the condition $r \notin \mathcal{F}(E)$ in the typing rule.

This proves that l is not contained in any value v' present in the store s' which implies that $\models s' : S'$. Thus, all the clauses of claim 3.47 are true and we have the desired result.

□

The soundness theorem immediately leads us to the following corollary that guarantees that closed reference locations are never updated.

Corollary 3.17 (Non-Mutability of Closed Locations) *Let a be an expression fragment within a type correct program p such that $E \vdash (\texttt{close}\ a) : \tau\ \mathit{ref}(\epsilon)$ and $e \vdash (\texttt{close}\ a)/s \Rightarrow l/s'$. Then, the location l is never updated during the evaluation of the rest of the program.*

Proof: The dynamic CLOSE rule (Figure 3.1) ensures that $\mathit{tag}(s'(l)) = \mathrm{ro}$. The ASSIGN rule requires a rw tag for the location to be updated, and there is no other rule that converts the tag of a location from ro to rw. Thus, as long as the program p does not illegally attempt to update the location l and runs into a dynamic error, the location l cannot be modified. This condition is guaranteed by the soundness theorem since the program is well-typed. □

Corollary 3.17 may be generalized to arbitrary objects with a completely closed type. This allows us to conclude that mutable objects, once successfully closed, can no longer be modified and therefore behave functionally.

Corollary 3.18 (Non-Mutability of Closed Objects) *Let a be an expression fragment within a type correct program p such that $E \vdash a : \tau$ where $\mathcal{R}(\tau) = \phi$ and $e \vdash a/s \Rightarrow v/s'$. Then, no location $l \in \mathit{Reachable}(v, s')$ is updated during the evaluation of the rest of the program.*

Proof: Using the soundness theorem we know that the evaluation of p (and hence a) does not lead to error and there exists a store typing $S' \models v : \tau$ and $\models s' : S'$. We claim that for all locations $l \in \mathit{Reachable}(v, s')$ we must have $\mathit{tag}(s'(l)) = \mathrm{ro}$. Otherwise, from Definition 3.12 Case 4 it follows that there exists a region variable r_1 such that $S'(l) = \tau_1\ \mathit{ref}(r_1)$. Then, using Proposition 3.15 it follows that $r_1 \in \mathcal{R}(\tau)$, which contradicts the hypothesis $\mathcal{R}(\tau) = \phi$.

Now, sound uses of the ASSIGN rule in Figure 3.1 require that the tag of the location being assigned should be rw. Furthermore, there is no rule that converts the tag of a location from ro to rw. Therefore, no assignments are possible on any location $l \in \mathit{Reachable}(v, s')$ during the evaluation of the rest of the program. □

Note that Corollary 3.17 is not a special case of Corollary 3.18 because Corollary 3.17 guarantees the non-mutability of a single closed location even if the locations reachable from within it are mutable. On the other hand, Corollary 3.18 only deals with objects that have completely closed types in order to guarantee that none of the locations reachable from them are mutable.

3.4 Type Inference

Finally, our type system admits a type inference algorithm *Infer* that infers principal types for expressions. This algorithm is a direct extension of the one described in Leroy's thesis [Ler92] to region variables. We only need to ensure that region variables are allowed to be unified only with other region variables and never with the null region (ϵ). This guarantees that we do not accidentally "close" a mutable reference type by unification. That operation should only be performed explicitly using the `close` construct.

We will not discuss the details of the inference algorithm here since it is a trivial extension of that in [Ler92]. We only state the following propositions that characterize the soundness and the completeness of the inference algorithm with respect to the type system described in Section 3.2:

Proposition 3.19 (Soundness of Type Inference) *Let a be an expression and E be a type environment. If $(\tau, \varphi) = Infer(a, E)$ is defined then we can derive $\varphi(E) \vdash a : \tau$.*

Proposition 3.20 (Completeness of Type Inference) *Let a be an expression and E be a type environment. If there exists a type τ' and a substitution φ' such that $\varphi'(E) \vdash a : \tau'$, then $(\tau, \varphi) = Infer(a, E)$ is defined and there exists a substitution ψ such that $\tau' = \psi(\tau)$ and $\varphi' = \psi \circ \varphi$.*

The proof of these proposition follows exactly as described in [Ler92].

Chapter 4

Closing Data-Structures

So far, we have shown how to close a single mutable reference location. In this chapter, we show how to extend the use of the `close` construct to complex, multi-level data-structures involving tuples, arrays, and general algebraic datatypes. First, we discuss some alternatives for specifying the dynamic and static semantics of closing multiple locations and regions simultaneously in a multi-level data-structure. This leads us to devise a type-annotation based specification mechanism within the source language that permits the user to specify exactly which regions and their corresponding locations are to be closed. Next, we discuss the strategies for verifying the correctness of this scheme for arrays and general algebraic datatypes. We also briefly discuss how this work may be applied to conventional languages such as C, Pascal, or Fortran. Finally, we present the summary of Part I and directions for future work based on this research.

4.1 Specification of "Close" for Multi-Level Data-Structures

The static and dynamic CLOSE rules shown in Chapter 3 (Figures 3.2 and 3.1 respectively) only apply to a single mutable location being returned as the only result from an expression. These rules clearly need to be extended for the diverse range of data-structures available in a modern programming language. Id offers tuples, arrays, and general algebraic datatypes (including recursive datatypes), any of which could be implemented in an imperative manner and may need to be closed. Furthermore, the exact mutable locations to be closed may be embedded anywhere inside a complex, structured result returned from a computation. Therefore, we need a systematic way of closing structured results which involves the following tasks:

1. Given an expression that returns a structured result, we need to specify which locations to close in the dynamic semantics, and the corresponding regions to close in the static semantics.

2. We need to statically verify the soundness of the `close` operation by clearly identifying the scope of the imperative operations taking place on the locations being closed.

As discussed in Section 2.3.1, treating the `close` construct as an *encapsulator* clearly delineates the scope of the imperative operations dynamically taking place on the returned result and it also statically identifies the type environment against which to verify the closing operation. In this section, we discuss the first issue of specifying the semantics of closing multiple locations and regions simultaneously within a structured result.

4.1.1 Dynamic Semantics Issues

A simple and natural way to extend the dynamic semantics of the `close` construct to multi-level data-structures is to take the "all-or-nothing" approach. That is, closing an arbitrary data-structure recursively closes all its subcomponents and failure to close any one of the sub-components results in the failure to close the entire data-structure. This generalized semantics may be expressed in the following dynamic rule for the `close` construct:

$$\text{DYNAMIC-CLOSE1:}\quad \frac{e \vdash a/s \Rightarrow v/s_1 \qquad \{l_1 \ldots l_n\} = \mathit{Reachable}(v, s_1) \qquad s_1(l_i) = v_i, \mathrm{rw} \quad 1 \leq i \leq n}{e \vdash (\mathtt{close}\ a)/s \Rightarrow v/(s_1 + \{l_i \mapsto v_i, \mathrm{ro}\}) \quad 1 \leq i \leq n}$$

In the light of the remarks made in Section 2.4.2, we have to be careful not to close locations that are reachable from the enclosing environment. Otherwise, we would be able to write a universal closing function such as the `closeall` function shown below that would incorrectly close arbitrary mutable objects that are still being used imperatively:

Example 4.1:

```
def closeall x = close x;
a = ref 1;
b = closeall a;
a := 2;                          % Dynamic Write Error!
```

Clearly, such functions should be disallowed because they create spurious dynamic "write-errors", *i.e.*, writing to a location that has been closed unintentionally. We would like to avoid such spurious errors or at least detect the possibility of creating such errors at the time of closing an object rather than at the time of using it. So we modify the DYNAMIC-CLOSE1 rule to reflect this strategy:

$$\text{DYNAMIC-CLOSE2:}\quad \frac{e \vdash a/s \Rightarrow v/s_1 \qquad \{l_1 \ldots l_n\} = \mathit{Reachable}(v, s_1) \setminus \mathit{Reachable}(e, s_1) \qquad s_1(l_i) = v_i, \mathrm{rw} \quad 1 \leq i \leq n}{e \vdash (\mathtt{close}\ a)/s \Rightarrow v/(s_1 + \{l_i \mapsto v_i, \mathrm{ro}\}) \quad 1 \leq i \leq n}$$

The above rule simply excludes all the locations reachable from the environment from being closed. This makes the `closeall` function of Example 4.1 behave like the `identity` function since no external location can now be closed. Alternately, we could introduce a side condition on the above rule to produce a dynamic "close-error" if any of the locations being closed was present in the environment.

The above rule is still not entirely free of spurious write-errors. In the light of the remarks made in Section 2.4.4, we should not close locations that are captured within a function closure because such locations may be modified by the function. The following example illustrates this scenario:

Example 4.2:

```
g = close { b = ref 1;
            def f x = { b := x; };
          in f };
...
g 2;                                  % Dynamic Write Error!
```

In the above example, an internal mutable location `b` is captured within a function closure `f` which is subsequently closed and returned. If the function body modifies the captured location (as it does here) then any application of the function would generate a spurious write-error. We can modify the DYNAMIC-CLOSE2 rule to omit closing such locations:

$$\text{DYNAMIC-CLOSE3:}\quad \frac{\begin{array}{c} e \vdash a/s \Rightarrow v/s_1 \\ \{l_1 \ldots l_n\} = \mathit{Closable}(v, s_1) \setminus \mathit{Reachable}(e, s_1) \\ s_1(l_i) = v_i, \mathrm{rw} \quad 1 \leq i \leq n \end{array}}{e \vdash (\mathtt{close}\ a)/s \Rightarrow v/(s_1 + \{l_i \mapsto v_i, \mathrm{ro}\}) \quad 1 \leq i \leq n}$$

The *closable* locations of a value v with respect to a store s, written $\mathit{Closable}(v, s)$, are defined to be all the reachable locations from the given value except those that are reachable *via* an embedded function closure. A simple way to compute this set would be to modify the algorithm GATHER-LOCATIONS given in Section 3.1.2 to collect the locations reachable through a function closure at Line 8 in a separate set. This set would then be subtracted from the set of all reachable locations of a value to yield the set of closable locations of that value.

The DYNAMIC-CLOSE3 rule given above seems fairly reasonable as far as the dynamic semantics of `close` is concerned for general, multi-level data-structures.

4.1.2 Static Semantics Issues

The static semantics for the DYNAMIC-CLOSE3 rule above could be given as follows:

$$\text{STATIC-CLOSE1:}\quad \frac{E \vdash a : \tau \qquad \{r_1 \ldots r_n\} = \mathcal{C}(\tau) \setminus \mathcal{F}(E)}{E \vdash (\mathtt{close}\ a) : \{r_i \mapsto \epsilon\}\tau \quad 1 \leq i \leq n}$$

The above rule erases only *closable* regions $\mathcal{C}(\tau)$ from the given type τ which consists of the set of all dangerous region variables of the given type except those that occur within the closure type of a function. It also excludes all regions visible in the type environment.

Although the rules DYNAMIC-CLOSE3 and STATIC-CLOSE1 seem plausible at first glance, unfortunately they cannot be shown to be sound with respect to each other. Intuitively, static semantics should provide a conservative approximation of what happens dynamically. As far as the `close` construct is concerned, this intuition is captured in Proposition 3.15 where we always maintain a correspondence between the reachable locations of a value and the visible regions variables in its type. Any semantics we give to the `close` construct must respect this correspondence, otherwise we will not be able to statically model the dynamics of closing an object properly. Unfortunately, the rules DYNAMIC-CLOSE3 and STATIC-CLOSE1 do not correspond to each other in this respect. Consider the following example:

Example 4.3:

```
x = ref (ref 1);
y = close { a = ref 2;                    % a ↦ l1
            b = ref 3;                    % b ↦ l2
            c = if true then a else b;    % l1 and l2 are region aliased.
            x := c;                       % l1 escapes.
           in b };
y := 4;                                   % Dynamic Write Error!
```

In the above example, `a` and `b` point to two independent reference locations, say l_1 and l_2. The conditional statement for `c` unifies the static region variables corresponding to these locations, therefore l_1 and l_2 become *region aliased*. This means that statically we cannot distinguish between these two locations. Assuming dynamically that the predicate resolves to `true` and `c` gets bound to l_1, we export l_1 into the environment by storing it into an external location and attempt to close l_2 by returning it as a closed result. Dynamically, l_2 is not visible in the environment so the DYNAMIC-CLOSE3 rule would close it. On the other hand, statically there is no difference between l_1 and l_2, and since l_1 is being exported, the static rule STATIC-CLOSE1 would not close the corresponding region variable creating a discrepancy between the static and the dynamic status of the location l_2. This would ultimately lead to a write-error on the l_2 location as shown.

Note that this write-error is generated not because the dynamic semantics for `close` was closing a location inappropriately as was the case for rules DYNAMIC-CLOSE1 and DYNAMIC-CLOSE2. This error came about because the static semantics was not sufficiently powerful to model the dynamic semantics accurately. One way to solve this problem is to classify such write-errors as static "close-errors" by making the static semantics little more conservative. This can be accomplished by causing the static rule to fail when a region variable cannot be closed rather than ignoring it. The following rule embodies this idea:

$$\text{STATIC-CLOSE2:} \qquad \frac{E \vdash a : \tau \qquad \{r_1 \ldots r_n\} = \mathcal{C}(\tau) \qquad r_i \notin \mathcal{F}(E) \quad 1 \leq i \leq n}{E \vdash (\texttt{close}\ a) : \{r_i \mapsto \epsilon\}\tau \quad 1 \leq i \leq n}$$

Using this rule, Example 4.3 would be classified as a static close-error and would be rejected, since an attempt was made to close a region (corresponding to locations l_1 and l_2) which could not be statically verified for correctness.

Unfortunately, the above rule still suffers from a rather technical problem that stems from our desire to perform type inference. It turns out that the above rule is not stable under type substitution (Proposition 3.9). In particular, the set of region variables $\mathcal{C}(\varphi(\tau))$ may turn out to be larger than the set $\varphi(\mathcal{C}(\tau)) = \{\varphi(r_1) \ldots \varphi(r_n)\}$ for a general substitution φ. This implies that new closable region variables may get introduced into a type by substitution that may not have been properly verified for correct `close` semantics previously.

Stability of substitution is used in showing semantic generalization (Proposition 3.14) as well as the soundness of type inference (Proposition 3.19). The former could be attributed to the specific style of relational semantics we have decided to follow in this thesis but the latter is fairly standard machinery in the literature and, if possible, we would like to retain it. Intuitively, failure of stability of substitution means that it may not be possible to show the soundness of a type inference algorithm based on this rule using standard unification and substitution machinery.

4.1.3 Combining Type Generalization and Closing

One way to devise a stable static rule for the `close` construct is to combine polymorphic generalization and object closing into a new language construct `letclose` $x = a_1$ `in` a_2 that behaves exactly like `let` $x = a_1$ `in` a_2 except that it erases all closable regions in the type of the expression a_1 and then immediately generalizes that type before binding the resulting type scheme to x. Intuitively, type generalization *protects* a typing derivation from later substitutions by quantifying its free type variables. Subsequent substitutions are then applied to polymorphic instantiations of the resulting type scheme which does not affect the original typing derivation.

A possible dynamic and static semantics of the `letclose` construct is shown below:

DYNAMIC-LETCLOSE:

$$\frac{\begin{array}{c} e \vdash a_1/s \Rightarrow v_1/s_1 \quad \{l_1 \ldots l_n\} = \mathit{Closable}(v, s_1) \setminus \mathit{Reachable}(e, s_1) \\ s_1(l_i) = v_i, \mathrm{rw} \quad s'_1 = s_1 + \{l_i \mapsto v_i, \mathrm{ro}\} \quad 1 \leq i \leq n \\ e + \{x \mapsto v_1\} \vdash a_2/s'_1 \Rightarrow v_2/s_2 \end{array}}{e \vdash (\texttt{letclose}\ x = a_1\ \texttt{in}\ a_2)/s \Rightarrow v_2/s_2}$$

STATIC-LETCLOSE:

$$\frac{E \vdash a_1 : \tau_1 \qquad E + \{x \mapsto \mathit{GenClose}(E, \tau_1)\} \vdash a_2 : \tau_2}{E \vdash \texttt{letclose}\ x = a_1\ \texttt{in}\ a_2 : \tau_2}$$

Where,

$$\begin{array}{rcl} \{r_1 \ldots r_n\} & = & \mathcal{C}(\tau) \setminus \mathcal{F}(E) \\ \tau' & = & \{r_i \mapsto \epsilon\}\tau \quad 1 \leq i \leq n \\ \{\alpha_1 \ldots \alpha_m\} & = & \mathcal{F}(\tau') \setminus \mathcal{D}(\tau') \setminus \mathcal{F}(E) \\ \mathit{GenClose}(E, \tau) & = & \forall \alpha_1 \ldots \alpha_m .\ \tau' \end{array}$$

These rules formalize what we have informally stated in the above paragraph. In this formulation, closing an object does not fail, instead, the definition of *GenClose* given above simply ignores such non-closable regions and does not generalize them. This property stems from the desire to keep type generalization as a non-failing property: if the type of an object cannot be generalized at a given scope, it is best left as a monomorphic type rather than flagging a "polymorphism-error".

Unfortunately, the above formulation suffers from the same region aliasing problem as discussed earlier in the context of the STATIC-CLOSE1 rule. Dynamically closable locations may be aliased to statically non-closable regions, and this discrepancy is silently ignored in the above rules. We can fix this problem as in the case of rule STATIC-CLOSE2, by flagging a static close-error if we fail to close a region that we were expected to close. Unfortunately, this conflicts with the requirement of non-failing type generalization.

4.1.4 Discussion

We have seen above that the problem of devising a sound static and dynamic semantics for a `close` construct for multi-level data-structures and functions is sufficiently tricky and has many potentially conflicting requirements. This warrants a re-inspection of our approach towards this problem.

Extending the static and dynamic semantics of a language to handle additional complexity and/or language constructs must fulfill the following requirements:

1. The dynamic semantics of a new language construct should be able to accurately *model* what that construct is *intended* to do in a simple and intuitive manner. The semantics should also take into consideration what is efficiently *implementable* on a machine. This conflict among what we intend, what we can model, and what we can efficiently implement is very important to resolve in the design of a new language construct.

2. Similarly, the static semantics machinery should be intuitive, efficiently implementable, internally stable, and externally consistent with respect to the dynamic semantics. The consistency requirement places a lot of constraints on the static machinery and it may not always yield the most general solutions.

3. Finally, we should also pay attention to other requirements on the design of a new language construct such as simple and understandable syntax, type inference etc. that may

not directly affect its semantics or the efficiency of implementation but may affect its widespread acceptability as a useful construct.

In the light of the above remarks, we have decided to abandon the search for a universal CLOSE rule. Below, we present our proposal for a family of CLOSE rules for closing a fixed set of regions and locations depending on the structure of the object at hand.

4.1.5 Closing a Fixed Set of Regions/Locations

The important point to realize is that closing a *known* set of locations that are characterized by a *statically fixed* set of region variables is perfectly sound. In the above examples, we ran into trouble when we tried to close an *arbitrary* set of locations for which we could not determine a *statically fixed* set of region variables.

In some sense, closing only a fixed set of region variables at a time gives us more fine grain control over what locations are being closed dynamically. In order for this strategy to work with multi-level data-structures, the following requirements must be met:

1. We need to specify statically which region variables we want to close.
2. We should be able to verify the soundness of closing these region variables against the type environment and other region variables that have not been closed.
3. The locations corresponding to the regions being closed must be similarly identifiable and closable in the dynamic semantics.
4. Finally, all the locations and the regions being closed and those that are left aside must jointly satisfy the region abstraction Proposition 3.15, *i.e.*, we cannot close a region variable statically without closing all its corresponding locations in the dynamic semantics and vice versa (region aliasing).

The above requirements directly lead us to an approach where we do not have universal static and dynamic semantics rules for the `close` construct. Instead, we have an algorithm to synthesize an exact static and dynamic semantics rule for each multi-level data-structure pattern that we wish to close. This would give rise to a family of rules depending on the structure of object at hand and the particular set of locations we wish to close within that object. For example, closing a n-tuple consisting of n reference locations can be accomplished using the following rules (*c.f.* single reference CLOSE rules in Figures 3.1 and 3.2):

$$\text{DYNAMIC-TUPCLOSE:}\quad \frac{\begin{array}{c} e \vdash a/s \Rightarrow \langle \texttt{n-tup}\ l_1 \ldots l_n \rangle / s_1 \qquad s_1(l_i) = v_i, \mathrm{rw} \quad 1 \leq i \leq n \\ L = \mathit{Reachable}(\langle \texttt{n-tup}\ l_1 \ldots l_n \rangle, s_1) \cup \mathit{Reachable}(e, s_1) \cup \\ \bigcup_{l' \in \mathit{Dom}(s)} \mathit{Reachable}(l', s_1) \end{array}}{e \vdash (\texttt{close}\ a)/s \Rightarrow \langle \texttt{n-tup}\ l_1 \ldots l_n \rangle / (s_1 \mid_L + \{l_i \mapsto v_i, \mathrm{ro}\}) \quad 1 \leq i \leq n}$$

$$\text{STATIC-TUPCLOSE:}\quad \frac{\begin{array}{c} E \vdash a : (\tau_1\ \mathit{ref}(r_1)), \ldots, (\tau_n\ \mathit{ref}(r_n)) \\ r_i \notin (\mathcal{F}(E) \cup \mathcal{F}(\tau_1) \cup \cdots \cup \mathcal{F}(\tau_n)) \quad 1 \leq i \leq n \end{array}}{E \vdash \texttt{close}\ a : (\tau_1\ \mathit{ref}(\epsilon)), \ldots, (\tau_n\ \mathit{ref}(\epsilon))}$$

Similar rules may be constructed for any subset of tuple fields containing reference values. Extending the above rules for closing tuples of references and vectors, we can easily handle the following example that combines their use in a non-standard way:

Example 4.4:

```
def polar2rect n =
   close { xs = i_vector (1,n);
           ys = i_vector (1,n);
           rsum = ref 0.0;
           _ = { for i <- 1 to n do
                   rad,theta = ... some large computation ...;
                   xs[i] = rad * sin theta;
                   ys[i] = rad * cos theta;
                   rsum := !rsum + rad;  }
          in !rsum/n, xs, ys };
```

Here, two vectors are closed and returned along with the accumulated average of a third quantity, all arising out of the same large shared computation. It is important not to repeat the computation and keep the storage space to a minimum. The use of an imperative style protected by the `close` construct makes the computation efficient and understandable without sacrificing overall functional behavior.

Steps in Synthesizing CLOSE Rules

In general, given an arbitrary program expression a that returns a structured result, synthesizing a specialized static CLOSE rule involves the following steps:

1. A group of region variables to be closed are identified from the type of the expression a using some appropriate language syntax.
2. These region variables are then verified for soundness. This requires that none of these region variables should occur in the type environment and in the type of the closed result being returned. Furthermore, none of these region variables should occur inside the closure type of an embedded function type as pointed out earlier.
3. If all the region variables pass the verification, they are erased from the type of the result, and the closed type is returned. Otherwise a static close-error is flagged.

Similarly, synthesizing a specialized dynamic CLOSE rule involves the following steps:

1. A group of locations to be closed is identified from the given value that correspond to the static region variables being closed.
2. These location are verified for possessing the read/write tag within the current store. Otherwise, a dynamic close-error is raised.
3. If all the locations pass the verification, their tags are flipped to read-only and the closed value is returned along with the current store with a slightly restricted domain as shown in Chapter 3 dropping any region-aliased handles to the locations being closed.

4.1.6 Type Annotations as "Close" Specifications

A simple way of specifying which regions to close in an arbitrary expression is to match it against a separate pattern and mark certain regions to be closed in that pattern. Note that this pattern matches the *type* of the expression and not its *value*. This is because several locations may be aliased to the same region variable by definition and we must close all of them simultaneously. Then, it makes sense to specify them once using their type rather than specify each of the locations individually.

A type pattern may be specified in a *type annotation* for the `close` expression as shown below:

EXPRESSIONS:	a	$::=$	$\ldots$	
		$\mid$	$(\texttt{close}\ a) :: \tau_{ann}$	`close` expression

Here, the expression a would usually be a program block which returns a structured result. The annotation type τ_{ann} would explicitly show the various type constructors present within the expression's type along with their region parameters. The precise regions parameters to be closed are specified using the null region (ϵ). The syntax used for specifying the annotation type is the full type grammar shown in Section 3.2.1 with the addition of a "don't care" type pattern (_) that may be used in place of any type, region, or closure type expression within the annotation. The scope of the free type, region, and closure extension variables of the annotation type is taken to be that annotation itself; annotation types in different parts of the program do not share variables.

Examples of this specification have already appeared in Chapter 2 within Examples 2.15, 2.20, and 2.21. The static typing rule for such type-annotated expressions may now be given as follows:

$$\text{ANNOTE-CLOSE:}\quad \frac{E \vdash a : \tau_{inf} \quad \{r_1 \ldots r_n\} = (\tau_{inf} \sim \tau_{ann}) \quad r_i \notin (\mathcal{F}(E) \cup \tau_{ann}) \quad 1 \leq i \leq n}{E \vdash (\texttt{close}\ a :: \tau_{ann}) : \tau_{ann}}$$

The type τ_{inf} stands for the inferred type of the expression a. The operation $(\tau_{inf} \sim \tau_{ann})$ matches the annotation type against the inferred type to determine the exact set of region variables being closed. Unlike the STATIC-CLOSE2 rule, this set remains stable under type substitution because the annotation type never changes. Below, we outline the mechanism of type and region matching and the subsequent verification of the `close` operation:

1. The types τ_{inf} and τ_{ann} must match exactly[1] except that some region variables in τ_{inf} may be closed in τ_{ann}. For each parameterized type constructor $T(\rho_1 \ldots \rho_n)$ the number of regions in the inferred and annotated type must also match. For syntactic convenience, we may allow a parameterized type constructor to appear without any region parameters in the given annotation, in which case all its region parameters are assumed to be the null region.

2. Each inferred region parameter is positionally matched with the corresponding annotated region parameter in order to determine the precise set of region variables being closed:
   - A null region in the inferred type must match a null region in the annotation type. These represent previously closed regions that cannot be opened again.
   - A region variable r in the inferred type matches a null region in the annotation type and is considered as *being closed* unless it occurs within the closure type of a function (Section 2.4.4). In the latter case, a static close-error is flagged.
   - A region variable r in the inferred type also matches a region variable r' in the annotation type as long as all occurrences of r in the inferred type match the same region

[1] Each occurrence of the "don't care" type pattern (_) within the annotation type is always assumed to match the corresponding type, region, or closure type expression present in the inferred type.

variable r' in the annotation type. For convenience, we may allow this matching to behave like a region variable constraint on the inferred region parameters rather than a mere renaming of variables. A unification substitution $\{r \mapsto r'\}$ may need to be generated in this case.

3. Finally, all region variables determined as being closed are collected in a set taking region variable constraints and variable renaming into account. This set of region variables, say $\{r_1 \ldots r_n\}$, can then be verified for soundness as shown in the above rule ANNOTE-CLOSE. Checking that no region variable r_i being closed appears anywhere within the current type environment E or within the annotation type τ_{ann} ensures that the corresponding closed locations are not reachable from the dynamic environment or the returned value. This is similar in spirit to the simple CLOSE rule shown in Figure 3.2.

The above scheme achieves both our original goals of specifying the regions to be closed and pinpointing the type environment to verify them against with a single, familiar language construct. Moreover, it specifies multiple regions to be closed at various levels of a structured result simultaneously, and it does this without adding additional semantic or syntactic complexity than was already present in the kernel language of Chapter 3.

This scheme also identifies the dynamic locations to be closed quite easily. The structure of tuple types directly reflects the structure of the tuples themselves. Therefore, the static distribution of regions variables to be closed within a structured type annotation directly leads us to the locations that need to be closed in the corresponding structured result. Locations within embedded function closures must never be closed, which is why the corresponding region variables are caught and flagged as a static close-error.

In the next two sections, we describe the semantics and `close` specification for arrays and general algebraic datatypes based on the above strategy.

4.2 Closing Arrays

4.2.1 Dynamic Semantics

We can easily generalize a single mutable reference location introduced in Chapter 3 to an array of *indexed* locations all of which belong to the same region. In fact, the `ref` construct may be viewed as a special case of a 1-dimensional array with length 1. Indexed locations effectively model consecutive memory addresses on which index computations may be performed, although the starting location of the array would still remain abstract. This treatment of locations is a little more concrete than that in Chapter 3 where every location was considered to be an independent abstract label.

We represent a 1-dimensional array as a pair $\langle \mathsf{vect}\ l, \underline{n} \rangle$ giving the starting location l and its length as a positive integer literal n. These are added to the set of dynamic values:

$$\begin{array}{llll} \text{VALUES:} & v & ::= & \ldots \\ & & | & \langle \mathsf{vect}\ l, \underline{n} \rangle \qquad \text{vector of length } n \end{array}$$

The values associated with the slots $0 \leq i < n$ of a vector $\langle \mathsf{vect}\ l, \underline{n} \rangle$ are stored at the locations $l, \ldots, l+n-1$ within the store s. All these locations are assumed to be directly accessible from the vector value:

$$\mathcal{L}(\langle \mathsf{vect}\ l, \underline{n} \rangle) = \{l, \ldots, l+n-1\}$$

$$\text{VECT-ALLOC:}\quad \frac{e \vdash a/s \Rightarrow \underline{n}/s_1 \qquad (l+i) \notin \mathit{Dom}(s_1) \quad 0 \le i < n}{e \vdash \texttt{allocvect}(a)/s \Rightarrow \langle \mathsf{vect}\ l, \underline{n}\rangle/(s_1 + \{l+i \mapsto \bot, \mathrm{rw}\}) \quad 0 \le i < n}$$

$$\text{VECT-DEREF:}\quad \frac{\begin{array}{c} e \vdash a_1/s \Rightarrow \langle \mathsf{vect}\ l, \underline{n}\rangle/s_1 \qquad e \vdash a_2/s_1 \Rightarrow \underline{i}/s_2 \\ (l+i) \in \mathit{Dom}(s_2) \qquad \mathit{value}(s_2(l+i)) = v \end{array}}{e \vdash a_1[a_2]/s \Rightarrow v/s_2}$$

$$\text{VECT-ASSIGN:}\quad \frac{\begin{array}{c} e \vdash a_1/s \Rightarrow \langle \mathsf{vect}\ l, \underline{n}\rangle/s_1 \quad e \vdash a_2/s_1 \Rightarrow \underline{i}/s_2 \quad e \vdash a_3/s_2 \Rightarrow v/s_3 \\ (l+i) \in \mathit{Dom}(s_3) \qquad \mathit{tag}(s_3(l+i)) = \mathrm{rw} \end{array}}{e \vdash (a_1[a_2] = a_3)/s \Rightarrow \texttt{()}/(s_3 + \{l+i \mapsto v, \mathrm{rw}\})}$$

$$\text{VECT-CLOSE:}\quad \frac{\begin{array}{c} e \vdash a/s \Rightarrow \langle \mathsf{vect}\ l, \underline{n}\rangle/s_1 \qquad s_1(l+i) = v_i, \mathrm{rw} \quad 0 \le i < n \\ L = \mathit{Reachable}(\langle \mathsf{vect}\ l, \underline{n}\rangle, s_1) \cup \mathit{Reachable}(e, s_1) \cup \\ \bigcup_{l' \in \mathit{Dom}(s)} \mathit{Reachable}(l', s_1) \end{array}}{e \vdash (\texttt{close}\ a)/s \Rightarrow \langle \mathsf{vect}\ l, \underline{n}\rangle/(s_1 |_L + \{l+i \mapsto v_i, \mathrm{ro}\}) \quad 0 \le i < n}$$

Figure 4.1: Dynamic Semantics of Arrays.

We also extend reachability (Definition 3.2) for vector values:

$$\begin{array}{lcll} \mathit{Reachable}(\langle \mathsf{vect}\ l, \underline{n}\rangle, s) & = & \phi & l \notin \mathit{Dom}(s) \\ \mathit{Reachable}(\langle \mathsf{vect}\ l, \underline{n}\rangle, s) & = & \mathcal{L}(\langle \mathsf{vect}\ l, \underline{n}\rangle) \cup \bigcup_{0 \le i < n} \mathit{Reachable}(\mathit{value}(s(l+i)), s) & \text{Otherwise} \end{array}$$

The algorithm Gather-Locations is correspondingly extended to collect such locations.

Figure 4.1 shows the dynamic semantics rules for 1-dimensional arrays. These are straightforward generalization of the corresponding rules for the `ref` construct. The primitive operator rules for vector allocation (`allocvect`), vector dereference (`a[i]`), and vector assignment (`a[i]=v`) operate as expected. During vector allocation, n fresh locations are added to the domain of the store each of which is initialized to a special "undefined" constant ($\bot$).[2] The domain validity test in dereference and assignment rules simulates *bounds checking* because only the indices within the bounds $l \cdots l + n - 1$ would be present within the domain of the store for a given vector value $\langle \mathsf{vect}\ l, \underline{n}\rangle$. Finally, the VECT-CLOSE rule closes all the locations of the vector simultaneously.

Multi-dimensional arrays may be modeled in a similar fashion or may be linearized into 1-dimensional arrays. In the latter case, the linearized vector value may need to keep additional information to translate a multi-dimensional index into a linearized index.

4.2.2 Static Semantics

Since arrays are considered to be homogeneous data-structures, all values contained in it must have the same type and all its locations must belong to the same region. This means that a single region variable suffices to represent the imperative properties of the array. Therefore, a mutable vector containing values of type τ is typed as $(\tau\ \mathit{vector}(r))$ just like a mutable reference type $(\tau\ \mathit{ref}(r))$. The free and dangerous variables of the vector type are also computed just like those for a reference type.

[2]This formulation is useful for synchronized arrays (I-structures and M-structures); conventional unsynchronized arrays as shown here may in fact be initialized with any constant of the appropriate type.

The types of the primitive array operators are shown below:

$$\begin{array}{rcl}
\mathit{typeof}(\texttt{allocvect}) & = & \forall t,u,r.\ \mathit{int} \xrightarrow{\langle u\rangle} t\ \mathit{vector}(r) \\
\mathit{typeof}(\texttt{_[_]}_{\mathit{mutable}}) & = & \forall t,u,r.\ (t\ \mathit{vector}(r), \mathit{int}) \xrightarrow{\langle u\rangle} t \\
\mathit{typeof}(\texttt{_[_]}_{\mathit{non\text{-}mutable}}) & = & \forall t,u.\ (t\ \mathit{vector}(\epsilon), \mathit{int}) \xrightarrow{\langle u\rangle} t \\
\mathit{typeof}(\texttt{_[_]=_}) & = & \forall t,u,r.\ (t\ \mathit{vector}(r), \mathit{int}, t) \xrightarrow{\langle u\rangle} \mathit{unit}
\end{array}$$

The static semantics rule for closing arrays operates exactly like that for the `ref` construct and is shown below:

$$\text{VECT-CLOSE:} \qquad \frac{E \vdash a : \tau\ \mathit{vector}(r) \qquad r \notin (\mathcal{F}(E) \cup \mathcal{F}(\tau))}{E \vdash \texttt{close}\ a : \tau\ \mathit{vector}(\epsilon)}$$

All the proofs for the `ref` construct given in Sections 3.1 and 3.2 extend naturally to arrays since all the locations contained within an array are simply an extension of its starting location l. We never create "internal" pointers into the middle of an array and operate on individual locations of the array. For instance, all indexed references on vectors operate on the value $\langle \mathsf{vect}\ l, \underline{n}\rangle$ and an index offset i, $l+i$ by itself is not taken to be a valid value. For the purpose of reachability, this ensures that all locations of an array are always taken together in a group which is similar in spirit to the treatment of the `ref` construct.

4.2.3 Semantic Model and Soundness

The store typing S carries the type $(\tau\ \mathit{vector}(\rho))$ at every location of the vector just like it carries the full reference type at a `ref` allocated location. Thus, we can extend the semantic model (Definition 3.12) in the obvious manner:

Definition 4.1 (Extended Semantic Model) *Let s be a store, S be a store typing, e be an environment, E be a type environment, v be a value, τ be a type, and σ be a type scheme. Define the following relations:*

Case *1: $S \models v : \tau$ — ...*

SubCase *1.6: $S \models \langle \mathsf{vect}\ l, \underline{n}\rangle : \tau\ \mathit{vector}(r)$, if $(l+i) \in \mathit{Dom}(S)$ and $S(l+i) = \tau\ \mathit{vector}(r)$ for all $0 \le i < n$.*

SubCase *1.7: $S \models \langle \mathsf{vect}\ l, \underline{n}\rangle : \tau\ \mathit{vector}(\epsilon)$, if $(l+i) \in \mathit{Dom}(S)$ and $S(l+i) = \tau'$ for all $0 \le i < n$. Furthermore, there exists a substitution φ with $\mathit{Dom}(\varphi) \subseteq \mathcal{F}(\tau') \setminus \mathcal{D}(\tau')$ such that $\varphi(\tau') = \tau\ \mathit{ref}(\epsilon)$.*

Case *4: $\models s : S$ — ...*

SubCase *4.3: If $S(l) = \tau\ \mathit{vector}(r)$ then $s(l) = v, \mathrm{rw}$ and $S \models v : \tau$.*

SubCase *4.4: If $S(l) = \tau\ \mathit{vector}(\epsilon)$ then $s(l) = v, \mathrm{ro}$ and $S \models v : \tau$.*

Proofs for semantic soundness from Section 3.3 also extend naturally to vectors using this extended semantic model. A simple reference value l is replaced by a vector value $\langle \mathsf{vect}\ l, \underline{n}\rangle$ and statements about the store typing of that location $S(l)$ are replaced by those applying to the group of locations $S(l+i)$ for all $0 \le i < n$. Proofs that do not directly depend on structure of values or of evaluation rules such as the region abstraction Proposition 3.15 do not change at all.

The above machinery allows us to finally answer the problem we posed at the beginning of Section 2.1 about implementing functional arrays in Id. The solution proposed in Section 2.3 for implementing function `make_vector` (Example 2.13) can now be automatically verified for correctness by the type system and is reproduced below:

Example 4.5:

```
i_vector     :: ∀t, u, r.(int, int) -⟨u⟩→ (t vector(r))
make_vector :: ∀t, u.(int -⟨u⟩→ t) → (int, int) -⟨int -⟨u⟩→ t⟩→ (t vector(ε))
def make_vector f (l,u) =
 close { a = i_vector (l,u);
         _ = { for i <- l to u do
                 a[i] = f i };
       in a };
```

The `i_vector` primitive allocates an empty vector between bounds (l, u) and initializes it to contain the "undefined" value $(\perp)$ everywhere. The region variable in the type of the allocated vector shows that it is assignable. On the other hand, the null region (ϵ) in the type of the returned vector from `make_vector` shows that it has been safely closed into a functional vector.

4.2.4 Modeling I-Structure and M-Structure Arrays

Readers may have noticed that the above description only presents unsynchronized mutable arrays that are closed into unsynchronized functional arrays. A few words are appropriate here regarding the modeling of synchronized (I-structure and M-structure) arrays present in Id.

As discussed in Section 2.3.5, a mutable array may be implemented using any one of the three underlying memory access protocols: unsynchronized, I-structure, or M-structure (refer Figure 2.1). Similarly, a functional array may be implemented using one of the two protocols: unsynchronized, or I-structure. However, the static typing machinery presented above allows us to only distinguish between a single mutable vector type $vector(r)$ and its corresponding functional vector type $vector(\epsilon)$. It does not matter which underlying protocol each type represents as long as we use the appropriate kind of barrier during the `close` operation (see Section 2.3.5), and that objects belonging to the two types are represented in the same way. The latter condition is required so that the `close` construct can simply change the *view* of an object from mutable to functional without requiring any data layout conversion.

In a conventional language such as C or Fortran, with only one kind of memory access protocol (unsynchronized), the simple two-way classification described above is sufficient. However, in Id we use two memory access protocols: I-structure and M-structure, giving rise to two types of assignable arrays and one type of functional arrays. Since, in Id functional objects are also implemented using I-structures, it is natural to use the I-structure protocol for objects with either the assignable type $vector(r)$ or the functional type $vector(\epsilon)$. This way, the underlying data layout is guaranteed to be the same in the two cases and no barrier is needed during the corresponding `close` operation. This leaves us with the question of how to type M-structure arrays and close them into functional arrays. Below, we discuss some possibilities.

One possibility is to assign M-structure arrays a separate mutable type constructor, say $m\_vector(r)$, and then somehow convert the type constructor *m_vector* into *vector* when closing. Semantically, this is not very clean because it requires an additional type conversion during the `close` operation. Moreover, this scheme does not express the language constraint that the layout of M-structure and functional objects is expected be the same. That constraint is buried under the semantics of the type conversion operation from M-structure objects to functional objects, which is left unspecified. Unsuspecting compiler writers may choose different data representations for M-structure and functional objects which would make the `close` operation on M-structure objects incorrect (or extremely inefficient).

Another possibility is to expand our region algebra to accommodate two different kinds of mutable objects: I-structure and M-structure. This is easily accomplished by using two kinds of region variables: r^i denoting I-structure regions, and r^m denoting M-structure regions. No implicit conversions would be allowed between the two kinds of region variables *via* type substitution or instantiation. The `close` construct would be used to explicitly close either kind of region variable into a null region. It is easy to see that all the semantic machinery presented in Chapter 3 would extend trivially to this scheme.

Under this scheme, a single parameterized type constructor may be used to denote all three kinds of arrays: $vector(r^m)$ for M-structure arrays, $vector(r^i)$ for I-structure arrays, and $vector(\epsilon)$ for functional arrays. The uniform type constructor used in all cases denotes the language constraint that the underlying data layout should be the same in all three cases. This scheme clearly separates the semantic modeling of the layout of an object which is denoted by its type constructor, from the modeling of its mutability and synchronization properties which is denoted by its region parameters.

It is easy to see that the region algebra may be enriched even further in order to accommodate unsynchronized objects within the same framework. This ability provides a natural extension to our type system when adding unsynchronized objects to Id, or adding I-structure and M-structure objects to conventional languages such as C or Fortran.

4.3 Closing General Algebraic Datatypes

4.3.1 Specification Issues

General algebraic datatypes introduce yet another dimension in the syntactic specification of closable regions and locations. In this section, we informally present some of the issues *via* examples that are formalized in later sections.

Multiple Region Parameters

Consider the functional list datatype declaration shown below:

Example 4.6:

```
type list t = nil | cons t (list t);
```

There are two fields in the `cons` constructor, either or both of them could be made mutable and closed independently. When a field of a datatype becomes mutable, it has to be tagged with a region variable which is reflected in the datatype constructor as a region parameter (*e.g.*, the type constructors $ref(\rho)$, or $vector(\rho)$). There is some flexibility in deciding whether to add additional region parameters to a type constructor for each mutable field or tag several mutable fields with the same region variable.

One possibility is to always require the user to specify the distribution of region parameters explicitly. On the other hand, it may be possible for the compiler to automatically add the region parameters to a mutable datatype declaration according to some fixed strategy. The question of whether two mutable fields should be modeled using the same region variable or not depends on how the fields are manipulated and closed within the rest of the program, although a fixed, compile-time heuristic is probably more desirable. For instance, the compiler could simply assign a single region variable per datatype or it could determine the largest independent set of region variables that would characterize a given datatype, subject to recursive typing constraints. Thus, either of the following declarations for mutable lists would be acceptable, although each provides a different degree of flexibility and approximation:

Example 4.7:

```
type list(r) t = nil | cons (r)!t (r)!(list(r) t);

type list(r1,r2) t = nil | cons (r1)!t (r2)!(list(r1,r2) t);
```

In the above declarations, we have prefixed a region variable to each of the mutable fields.[3] The first declaration identifies the entire spine of the list with the same region, while the second declaration classifies heads and tails separately. Whether the first or the second declaration should be used depends on whether we wish to close heads of a list without closing the tails or vice-versa. In general, it is useful to have as much flexibility as possible, especially if the heads and tails employed different memory synchronization protocols (see Section 2.3.5), so the second declaration appears to be a better choice. However, note that both fields share the same type variable (t), so we will not be able to generalize objects of this list type unless both regions are closed. Therefore, if we are only concerned about converting mutable lists to completely functional lists, then collapsing the two regions into a single one may be more desirable since it simplifies the datatype representation.

Inherited Region Parameters

Embedded parameterized types within another algebraic datatype forces the type constructor being defined to inherit the region parameters of the embedded type, otherwise there would be no way to generalize such region variables in a Hindley/Milner type system. For example:

Example 4.8:

```
type keyref(r) t = mkkeyref I (ref(r) t);
```

Although, none of the fields of the type *keyref* itself is mutable, it still must inherit the region parameter r of the embedded type *ref*, otherwise this parameter could never be generalized and would always point to the same region. This information can easily be taken into account within the compiler while computing the region parameters of a datatype declaration automatically.

Closure Type Parameters

An interesting problem occurs with general algebraic datatypes that may hide function closures inside them. The closure typing system described in Section 2.2.6 works well with higher-order functions since we have a way of expressing, propagating, and generalizing over closure types directly as they are defined while typing a λ-abstraction or instantiated at a function reference. But, if a function is carried indirectly by storing it within a data-structure, we must still not lose its closure typing information because of such indirection. Otherwise, write handles embedded inside such functions could escape undetected. To illustrate this subtle point, consider the following example:

Example 4.9:

```
type capture t0 = capt (int -⟨u0⟩-> t0 -⟨u1⟩-> t0);

def escape_5 n =                    % escape_5 :: ∀t0.int → (vector t0, capture t0)
  close { a = i_vector (1,n);
          def g i v =               % g :: ∀u2 u3.int -⟨u2⟩-> t0 -⟨vector(r) t0, u3⟩-> t0
            { a[i] = v; in v };
```

[3]A *dot* (.) in front of a field denotes that it is an I-structure field, while a *bang* (!) denotes that it is an M-structure field.

```
in a, capt g } :: (vector _),_;
```

As shown above, the datatype `capture` has a single type parameter t_0 and its sole constructor `capt` stores a polymorphic function closure. It is necessary to parameterize this type with the closure extension variables u_0, u_1 of the hidden function type so that these variables can also participate in type generalization and `close` verification. With the declaration shown above, the type system will be unable to detect that a write handle to the array `a` being closed is escaping *via* a function closure since that function is hidden inside a data-structure. We should point out that this parameterization is necessary for the closure typing system itself to work properly, this is not specifically related to the `close` construct. Without such parameterizations one would be able to *launder* functions with complicated closure types by simply storing them into a data-structure and then fetching them back. The correct declaration for the datatype `capture` is shown below with additional closure type parameters:

Example 4.10:

`type capture` $t_0\ u_0\ u_1$ `= capt` $(int \prec\!\langle u_0\rangle\!\succ t_0 \prec\!\langle u_1\rangle\!\succ t_0)$;

The closure type parameters on datatypes behaves exactly like closure extension variables within closure types of function. For example, the type of `capt g` now instantiates the extension variable u_1 of the datatype `capture` with the closure type $(vector(r)\ t_0, u_3)$ of the function `g`, thereby exposing the hidden region r embedded within the closure type. This would allow the subsequent `close` verification process to flag the escaping region as a static close-error.

4.3.2 Syntactic Specification of Algebraic Datatypes

Now, we are ready to show the full machinery for the specification of general algebraic datatypes. A general algebraic datatype declaration is shown below:

$$\begin{array}{rclllll} \texttt{type}\, T(r_{1\ldots i})\, t_{1\ldots j}\, u_{1\ldots k} & = & C_1 & (\rho_{11})\tau_{11} & \ldots & (\rho_{1a_1})\tau_{1a_1} \\ & | & \vdots & \vdots & & \vdots \\ & | & C_m & (\rho_{m1})\tau_{m1} & \ldots & (\rho_{ma_m})\tau_{ma_m} \end{array}$$

This declares a type constructor T, with $r_i \ldots r_i$ as region variable parameters, $t_1 \ldots t_j$ as type variable parameters, and $u_1 \ldots u_k$ as closure extension variable parameters. This datatype has m constructor disjuncts $C_1 \ldots C_m$ each with its own arity $a_1 \ldots a_m$ any of which could be zero. Each field of a non-nullary constructor C_p has an independent type τ_{pq} and a region expression ρ_{pq}. The type τ_{pq} may use region, type, and closure variables from the declared parameters of the datatype T. The region expression ρ_{pq} either consists of exactly one region variable parameter denoting that this field is mutable or it is the null region ϵ denoting that this field is functional.[4]

The above declaration may be supplied by the user, or the compiler may automatically augment an ordinary datatype declaration containing only type variable parameters with additional region and closure extension parameters. In order to do so, the user must at least specify which fields are expected to be mutable and which ones are functional. Then, a *maximally independent* set of region variable parameters and a set of closure extension parameters may be computed for each datatype T declared within the program using the following steps:

1. First we assign region expressions ρ^T_{pq} to each field of each datatype T declared within the

[4]Additional syntax may be used to distinguish between I-structure and M-structure fields.

program as follows:

$$\rho^T_{pq} = \begin{cases} r_{pq} & r_{pq} \text{ is new and the } pq\text{-th field in the datatype } T \text{ is mutable} \\ \epsilon & \text{Otherwise} \end{cases}$$

Each datatype T is initially assigned the region parameters $R^T = \bigcup \rho^T_{pq}$ and the closure extension parameters $U^T = \bigcup \mathit{Closure\text{-}Variables}(\tau^T_{pq})$.

2. Now we construct a *datatype reference graph* consisting of all the datatypes declared within the program, where there is an edge from a datatype T_1 to another datatype T_2 if T_2 occurs within some field type τ_{pq} of T_1. We partition the nodes of this graph into *strongly connected components* (SCC) [AHU74] according to this (directed) edge criterion. This puts mutually recursive datatypes into the same component. We will use this information to assign the same region and closure extension parameters to mutually recursive datatypes.

3. Now, proceeding in a topologically bottom-up fashion on each SCC of the above reference graph, we compute the final set of region and closure extension parameters for each datatype as follows. If two datatypes T_1 and T_2 belong to the same SCC, then all occurrences of one inside the other use the same variables. If T_1 refers to T_2 and they belong to different SCCs, then for each occurrence of T_2 within the declaration of T_1 we rename the parameters associated with T_2 (R^{T_2} and U^{T_2}) to fresh variables and recompute the parameters of T_1 (R^{T_1} and U^{T_1}).

4. Finally, each datatype T within the same SCC is assigned the region parameters $\bigcup_{T \in SCC} R^T$ and the closure extension parameters $\bigcup_{T \in SCC} U^T$.

Intuitively, the above algorithm assigns a new region variable to each statically distinguishable mutable field keeping track of inherited and recursive regions. In this sense, it computes a maximally independent set of region variables for each datatype. For example, this algorithm would automatically compute the region assignment ($\mathit{list}(r_1, r_2)\ t$) shown in Example 4.7 for the following type declaration which specifies both heads and tails as being mutable:

Example 4.11:

```
type list t = nil | cons !t !(list t);
```

4.3.3 Dynamic Semantics

Dynamically, each constructor disjunct C_p gives rise to a value $\langle \mathsf{C}_p\ v_1 \cdots v_{a_p} \rangle$ where C_p denotes a tag that identifies the disjunct and $v_1 \cdots v_{a_p}$ are its field values. The value corresponding to a mutable field is a unique location l_{pq} whose contents are accessible through the store. This generalized representation subsumes the functional n-tuples ($\langle \textsf{n-tup}\ v_1, \ldots, v_n \rangle$) and single mutable reference cells (l) used in Chapter 3 because it permits individual locations of a tuple itself to be mutable. In order to avoid confusion, we now represent individual mutable reference cells such as those used in Chapter 3 using the following datatype declaration:

Example 4.12:

```
type ref(r) t = ref (r)!t;
```

A mutable reference cell would now be represented as $\langle \mathsf{ref}\ l \rangle$ instead of a bare location l which by itself is no longer considered to be a proper value and may only appear as a mutable field value within a constructor value.[5]

The locations directly contained in a constructor value $\mathcal{L}(\langle \mathsf{C}_p\ v_1 \cdots v_{a_p} \rangle)$ are naturally defined to be the set of field values that are locations. Similarly, the reachable locations of a constructor value (with respect to a store s) are the set of locations directly or indirectly reachable from all the fields of the constructor.

The primitive operations of allocation, dereference, and assignment extend naturally to constructor disjuncts and their embedded mutable and non-mutable fields. The reader is referred to [Nik91] for details of the exact syntax used in Id. The dynamic semantics of these operations is given by a family of allocation, dereference, and assignment rules on the lines of those shown for reference cells in Chapter 3.

The dynamic semantics of closing a constructor value follows the discussion in Section 4.1. The main problem is to identify the set of dynamic locations to match the specified region variables that are being closed in a general algebraic datatype. For non-recursive datatypes, the locations to be closed are exactly those carried directly within the constructor value at the field position corresponding to the region variable being closed. As an example, we reproduce the `point` datatype from Example 2.17 below with explicit region parameters. Both fields of the point `pt1` are closed while only the second field of `pt2` is closed:

Example 4.13:

`type point`(r_1, r_2) `= pt` (r_1)`!`*float* (r_2)`!`*float*`;`

`pt1 = close (pt 1.2 3.5) :: point;` % Abbreviation for `point`(ϵ, ϵ)
`pt2 = close (pt 2.2 4.7) :: point`$(\_, \epsilon)$`;`

For recursive datatypes, the value contained within each field that recursively refers to the region variable being closed must also be traversed and closed. Consider the following example using mutable lists:

Example 4.14:

`type list`(r_1, r_2) t `= nil | cons` (r_1)`!`t (r_2)`!(list`(r_1, r_2) t`);`

`l1 = close (1:2:3:4:nil) :: (list`$(\epsilon, \_)$ `int);`

The dynamic implication of closing the first region parameter r_1 of the list `l1` is that all head fields on the spine of the list get closed, although the tail fields still remain mutable (since r_2 is not closed). This is because after closing the head field of the first `cons`-cell, we must recursively traverse its tail field in order to close the region parameter r_1 in the remaining list. This process continues until we hit `nil` in the tail field since there are no more fields to recurse into.

Now, we show a real example involving recursive datatypes that shows the usefulness of the `close` construct in building functional objects from the corresponding mutable ones. We present an efficient implementation of the `map_list` function that does not even require reversing the final list (*c.f.* function `imp_map` in Example 2.6) because the list is generated from left to right using a technique known as "open-lists" [ANP89]:

[5] We abuse our notation slightly by calling locations embedded inside a constructor value as field values just like the other values present directly within the constructor, although bare locations are no longer considered to be proper values. They only serve to define the domain of the mutable store.

Example 4.15:

```
def map_list f nil =  nil
  | map_list f (x:xs) =
   close {
     hd = cons _ _;                % The expression (cons _ _) allocates a ⟨cons ⊥, ⊥⟩
     hd.cons_1 = f x;
     tl = hd;
     finaltl = { while not (nil? xs) do
                   newtl = cons _ _;
                   next x : next xs = xs;
                   newtl.cons_1 = f x;
                   tl.cons_2 = newtl;
                   next tl = newtl;
                 finally tl };
     finaltl.cons_2 = nil;         % The expression nil allocates a ⟨nil⟩
   in hd } :: (list _);            % Abbreviation for (list(ε, ε) _)
```

Finally, observe that the set of locations that need to be examined for closing a given region variable in a general algebraic datatype depends solely on its type declaration. For instance, we know at the time of declaring the `list` datatype (Example 4.14) that the region variable r_1 occurs inside the type of its tail field. Therefore, we need to examine all the `cons`-cells on the spine of the list in order to close the region variable r_1. But we do not have to examine the objects contained within the head fields in order to close the region r_1. If r_1 occurred inside the type of the objects contained within the head fields, then the static semantics for the `close` operation described below would generate a static close-error and such a program would be rejected. Thus, an exact dynamic CLOSE rule can always be constructed for each region variable of a polymorphic, user-defined datatype at the time that datatype is declared without regard to how it is instantiated at various places within a program.

4.3.4 Static Semantics

The free variables of a general algebraic datatype are defined as follows:

$$\mathcal{F}(T(\rho_{1\ldots i})\ \tau_{1\ldots j}\ \pi_{1\ldots k}) = \cup_i \mathcal{F}(\rho_i) \bigcup \cup_j \mathcal{F}(\tau_j) \bigcup \cup_k \mathcal{F}(\pi_k)$$

The dangerous variables of a general algebraic datatype may either be dangerous within one of its argument types τ_j or closure types π_k, or they may occur within the type of a mutable field of one of its constructors. In the latter case, all the type variable parameters occurring within that field are *inherently* dangerous much like the type of an object contained within a mutable reference cell. Therefore, we define:

$$\mathcal{D}(T(\rho_{1\ldots i})\ \tau_{1\ldots j}\ \pi_{1\ldots k}) = \cup_i \mathcal{F}(\rho_i) \bigcup \cup_k \mathcal{D}(\pi_k) \bigcup \cup_j \begin{cases} \mathcal{F}(\tau_j) & \text{If } t_j \text{ occurs inside a mutable field} \\ \mathcal{D}(\tau_j) & \text{Otherwise} \end{cases}$$

Finally, the dangerous region variables of a general algebraic datatype are defined as follows:

$$\mathcal{R}(T(\rho_{1\ldots i})\ \tau_{1\ldots j}\ \pi_{1\ldots k}) = \cup_i \mathcal{F}(\rho_i) \bigcup \cup_j \mathcal{R}(\tau_j) \bigcup \cup_k \mathcal{R}(\pi_k)$$

The types of the primitive operators for allocation, dereference, and assignment of constructors and their fields are defined as expected.

The static CLOSE rule also follows the discussion of Section 4.1. We only need to show how to perform the verification for flagging a static close-error for algebraic datatypes. This is done as follows:

1. Given an type-annotated expression, (`close` a) :: $T(\rho_{1\ldots i})\ \tau_{1\ldots j}\ \pi_{1\ldots k}$, along with an inferred type $T(\rho'_{1\ldots i})\ \tau'_{1\ldots j}\ \pi'_{1\ldots k}$, first we match the regions $\rho_{1\ldots i}$ specified in the annotation against the corresponding regions $\rho'_{1\ldots i}$ of the inferred type. Null regions in the inferred type must exactly match the corresponding regions in the annotation type. While some region variables in the inferred type may be constrained to be closed (mapped to ϵ), other region variables are simply renamed/unified to the region variable specified in the constraint.

2. The candidate region variables so determined to be closed, say $\{r_1 \ldots r_n\}$, must not occur inside a function closure type within the inferred type parameters $\tau'_{1\ldots j}$ or within the inferred closure parameters $\pi'_{1\ldots k}$. This ensures that we do not close region variables that are captured inside function closure types.

3. Finally, the region variables being closed must satisfy the following test with respect to the annotation type:

$$\forall r \in \{r_1 \ldots r_n\} \qquad r \notin \left[\mathcal{F}(E) \bigcup \cup_j \mathcal{F}(\tau_j) \bigcup \cup_k \mathcal{F}(\pi_k)\right]$$

If any of the above tests fails, we flag a static close-error. Otherwise, the `close` operation is considered to be successful.

4.3.5 Soundness

The static and dynamic CLOSE rules for general algebraic datatypes described above are direct extensions of the formal machinery shown for reference cells in Chapter 3. It is reasonably straightforward to see that we follow the same idea of specifying a fixed set of static regions to be closed for an identifiable set of dynamic locations. Therefore, all the semantic machinery given in Chapter 3 extends naturally to this framework.

4.4 Functional Encapsulation in Conventional Languages

We mentioned in Section 1.3 that the functional encapsulation mechanism presented in this thesis would also be quite useful in a monomorphic, first-order language such as C, Pascal, or Fortran. However, adding this mechanism to a conventional language may require a few changes in the language and its type system, a possible change in the programming style, as well as possible simplifications within the proposed type system itself. In this section, we outline how all this might be achieved using C as an example.

It is clear that in order to make any kind of guarantees based on the type system, we must have a strongly-typed language. C is not strongly-typed because it allows unrestricted type conversion among object at the discretion of the user *via* type-casting [KR88]. Using this facility the user may convert pointers to closable objects into non-pointer datatypes and vice-versa, thereby completely throwing off our type analysis. Therefore, no type-casting may be allowed in order to ensure sound, verifiable functional encapsulation.

The type system of C would obviously need to be extended with regions, although with suitably chosen syntactic defaults regions may not appear explicitly in many cases. For instance, the compiler may automatically assign region parameters to all `struct` and `union` type declarations as discussed in Section 4.3.2. The compiler would also need to define a unique memory allocation function for each declared datatype. This is necessary because, as discussed above, we have to eliminate the use of type-casts which is most often used to fix the type of a freshly allocated object using the only available memory allocation function `malloc`.

The most important simplification in our type mechanism would be that we would no longer need closure types. Although, C allows passing function pointers as arguments and results, functions are only declared at the top-level and they may only have free identifiers that are also declared at the top-level. Therefore, the types of such free identifiers would always be visible within the global type environment and can never be closed accidentally. In other words, we do not need to keep track of the types of the free identifiers of a function because such types would always be present in its enclosing type environment anyway.[6] This greatly simplifies our typing machinery and makes its even more intuitive and easy to use.

Finally, we must point out that functional encapsulation is useful only if we localize the allocation and construction of objects to nested program blocks. This facility encourages a programming style where we dynamically allocate and update an object in a deeply nested block, and then close and return that object into the enclosing block where it may be used functionally. This style is certainly possible in C and Pascal but may preclude some earlier versions of Fortran due to the lack of block-structure and dynamic memory allocation.

4.5 Conclusions

4.5.1 Summary of Part I

In the preceding chapters we have presented a powerful type system that fulfills our goal for providing a sound and verifiable type abstraction mechanism between the high-level functional layer and the low-level imperative layer of a polymorphic programming language. We started with the problem of implementing functional array constructs present in our high-level language in terms of low-level imperative program fragments written in a small kernel language without sacrificing storage efficiency or parallelism. In the process, we introduced a new construct within the kernel language called "`close`" that changes the *view* of a mutable data-structure from imperative to a functional one. The type system statically verifies the soundness of such a change and guarantees that successfully closed objects are never updated again during execution.

We also showed how to extend the use of the `close` construct to complex data-structures within the language including arrays, tuples, functions, and general algebraic datatypes. We discussed issues of language design and specification of closing such data-structures and its effect on other language features such as type polymorphism and dynamic memory synchronization protocols. Our proposal for syntactically specifying closable objects blends nicely with already existing mechanisms of specifying type declarations and type annotations for program expressions.

The type abstraction mechanism described in this thesis helps both compiler and language designers as well as the end-users. On the one hand, it helps to reduce the size of the compiler by permitting efficient implementations of high-level, functional constructs (*e.g.*, `make_vector` in Example 4.5 and `map_list` in Example 4.15) to be pushed into system libraries rather than being implemented within the compiler as primitives. On the other hand, it provides a tool

[6] This is also true in Pascal and Fortran even though Pascal allows internal function declarations [JW75]. This is because in all these languages functions are never passed outside the scope of their definitions.

for the end-user to design arbitrary new functional data-structures more efficiently using imperative kernel constructs and then safely close them (*e.g.*, `histogram` in Example 2.16 and `polar2rect` in Example 4.4). In this sense, our type system provides a safe and controlled abstraction mechanism for the end-user to exploit the power and efficiency of low-level, imperative constructs without destroying the clean semantics of high-level constructs.

4.5.2 Implementation Status

The type system described in this thesis is currently unimplemented. Therefore, our claims of displacing wired-in implementation of functional data constructors within the Id compiler in favor of system libraries, and user-level flexibility in implementing new functional abstractions are yet to be tested. Currently, the Id compiler uses several internal "hacks" to provide these functional abstractions which would clearly be unsound if exposed to the user directly.[7] Our typing machinery would have the effect of cleaning and legitimizing these hacks into proper kernel language features. Our type system would also combine three different type declarations used for M-structure, I-structure, and functional data objects into a single declaration as discussed in Section 4.2.4.

Currently, the Id language is undergoing major revisions and in its next incarnation as pH [NAH93] we hope to include some of the ideas embodied in this thesis.

4.5.3 Future Work

As mentioned above, the obvious first task for us is to implement this type system fully and study its usefulness not only in terms of the semantic cleanliness but also its implementation efficiency and ease of use. We would like to implement this system both for Id (and pH) as well as a restricted subset of the C language as outlined in Section 4.4. Below, we discuss some alternate directions for future research.

Theoretical Improvements

There are several aspects of the current research that need more detailed scrutiny. Throughout in this thesis, we have used a strict, sequential dynamic semantics for our kernel language. We were able to do this because the problem of closing imperative data-structures is largely orthogonal to the issues of parallelism and synchronization which would have only made the formalization of the soundness proofs much harder. But it would be useful to show the soundness proofs directly in a parallel setting. This would also allow us to directly model the different closing strategies required with different memory synchronization protocols as discussed in Section 2.3.5. We feel that a graph rewriting framework such as [AA93] would be more appropriate for this purpose than the relational semantics approach taken here.

Applications to Other Compiler Analyses

This type system may also be used to infer useful static information that is conventionally determined using dataflow analysis or abstract interpretation. For example, we know that the static verification strategy for the `close` construct provides a limited form of object escape analysis. It guarantees that there are no additional references to the object being closed other

[7]The current version of the Id compiler uses `typeconverter` declarations that simply change the type of an object without any semantic verification. It also uses internal *pragmas* to "fix" the functional polymorphism of array and list comprehension desugaring.

than the reference being returned from the `close` expression. This implies that the enclosing program fragment that receives the closed object has *exclusive* access to that object. If we do not make the object read-only upon closing, then this type mechanism effectively provides a static way for verifying exclusive dynamic access to a mutable object without using any synchronization primitives (such as semaphores) or single-threading the object through the entire program. The enclosing program fragment could make exclusive, unsynchronized read/write accesses to the object for some time then pass out multiple references to other sub-programs. All such references may again be brought together and again checked for escape in an enclosing scope.

Another important observation is the dynamic life-time of an object that is shown to be closable at the boundary of a `close` expression and is actually not returned from that expression, is guaranteed to be bound to the scope of that `close` expression. This is because no references to that object may escape this scope. This information may be used to allocate such objects on stack instead of the heap as shown in [TT93], or insert additional code at compile-time to reclaim that storage automatically on the lines of [HJ92].

Part II

Types in Run-time System Design: Type Reconstruction

Chapter 5

A Typed Run-time System

5.1 Introduction

Traditionally, programming environments of dynamically-typed languages such as Lisp or Smalltalk maintain type information in the form of run-time type descriptors on every object. This information may be used, for instance, to detect run-time type-errors, to dispatch to different handlers for a given operation based on the type of the arguments, and to distinguish pointer data from non-pointer data for the purpose of garbage collection. Although very flexible in design, such language implementations pay the price of managing type-tags either in the form of complex specialized hardware or in the form of extra space and time requirements in software.

Languages geared towards high performance computation such as C or Fortran take the other extreme. They aim for a very simple and efficient run-time system with no type information to be maintained at run-time. The user is made directly responsible for complex tasks that may require run-time type information such as ensuring type consistency and automatic storage management. If necessary, the compilers for these languages can be explicitly instructed to generate static type information to be used for specific run-time applications such as source-level debugging.

Several important questions arise at this point. What is the advantage of having type information available at run-time? What specific applications may use run-time type information? How much type information is desired, complete source-level types or a partial specification? What language design features may help or complicate the task of making run-time type information available? How much of this type information can be pre-computed by the compiler and how? Do we need to carry the type information throughout execution or can it be reconstructed on demand? What is the run-time cost of such type maintenance or reconstruction? And finally, how does a typed language and its run-time system compare in terms of overall performance, program reliability, and user flexibility to other systems?

In Part II of this thesis, we attempt to answer some of the above questions in the context of the Id programming language and its run-time environment. We study how source-level type information can be propagated through the compiler and made available during the execution of a program. We also discuss specific applications that use this information at run-time.

5.2 Design Issues for a Typed Run-time System

Several language design features affect the availability and the accuracy of type information during the execution of a program. Likewise, run-time system design decisions affect the overall

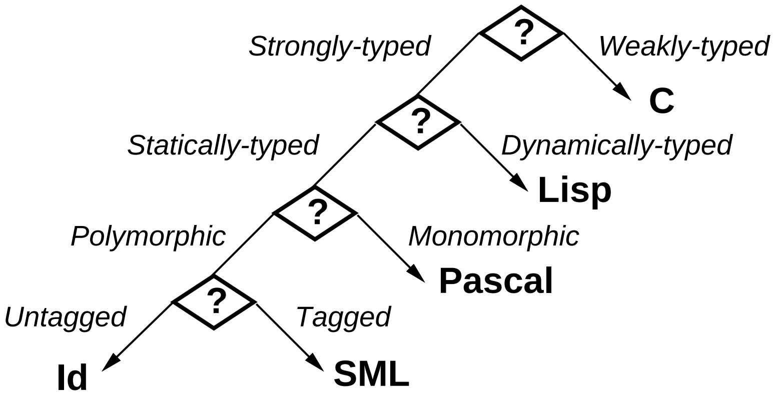

Figure 5.1: Design Issues for a Typed Run-time System.

cost of computing and propagating this type information. Figure 5.1 shows several such design issues and classifies some existing programming languages on their basis. We discuss these issues below.

5.2.1 Strong *vs.* Weak Typing

Strongly-typed languages such as Pascal, Lisp, or Standard ML provide a consistent model of assigning a type to every data object and every sub-computation in a program. Computations are allowed to proceed only if provided with objects of the right type. Enforcing type consistency allows run-time type information to serve as a reliable description of the computation being performed at any time. Therefore, it makes sense to use this information, if available, for applications that operate on a wide variety of run-time data and need some mechanism to identify and distinguish among them. Applications such as displaying objects in a source debugger, marking objects in a garbage collector, and object I/O fall into this category.

Weakly-typed languages such as C or Fortran permit the user to arbitrarily coerce the type of an object to another type. This makes the currently assigned type of an object to be a poor description of its actual contents. It is still possible to view an object according to its currently assigned type, but there is no guarantee that it provides the complete and accurate description of the object. Therefore, providing reliable type information at run-time is possible only in a strongly-typed system.

5.2.2 Static *vs.* Dynamic Typing

Compilers for statically-typed languages such as Pascal, or Standard ML enforce the type consistency expected from a strongly-typed program at compile-time. This frees up the system from the responsibility of checking for type consistency at run-time. Some modern languages like Haskell also provide systematic mechanisms to resolve overloading of operators and selection of methods at compile-time based on the static types of their arguments [WB89]. Therefore, static typing offers many of the advantages of dynamic availability of type information without actually carrying that information at run-time. Moreover, all the static type information may be saved and used in optimizations during the compilation phase itself or in other run-time applications during program execution. Although, additional work may be needed to reproduce the desired information at run-time when demanded.

5.2.3 Tagged *vs.* Untagged Object Model

A simple way to provide type information at run-time is to tag every object: a few bits (usually one or two) in every word may be used as a *tag* to distinguish scalar objects from pointers to heap objects. More information about the type and size of objects may be kept in an object header. All dynamically-typed languages such as Lisp and Smalltalk use extensive tagging of objects in order to perform type consistency checks at run-time. Some implementations of statically-typed languages such as the Standard ML of New Jersey [App90] also make use of object tagging, usually for the benefit of the garbage collector.

Tagging every object is costly. Keeping tag bits in every word reduces the range of representable scalars and pointers in conventional architectures, and the user application also pays the additional cost of tag maintenance. Sometimes, scalar values (usually floating point numbers) may be boxed in a heap data-structure in order to preserve their full range. This incurs the additional cost of allocating the box and accessing it indirectly.

Keeping objects untagged simplifies the memory model and eliminates the space and time overheads, but no type information is directly available at run-time. In weakly-typed languages such as C or Fortran, the user is directly held responsible for generating and propagating consistent type information at run-time. In statically-typed languages such as Pascal or Id, the compiler and the run-time system may share the responsibility for carrying the type information. The compiler may generate detailed symbol tables for each function in the program. The run-time system may load and process the information before program execution or upon request from another application.

5.2.4 Type Maintenance *vs.* Type Reconstruction

Recently, several **type reconstruction** schemes have been proposed for statically-typed polymorphic languages that do not incur the run-time tag management overhead [App89, Gol91, GG92]. In these schemes, static type information may be combined with clues from the dynamic state of the machine (the call stack) to automatically reconstruct the run-time type of most run-time objects. Therefore, with a small cost of type reconstruction, the type-tags on such objects may be safely dropped without compromising the ability to determine their exact run-time types.

If the semantics of a language necessitates a tagged or boxed representation for objects, or if special hardware support for tags is available, then run-time type reconstruction is probably not the right choice. For example, compiler-directed type reconstruction is impossible in a dynamically-typed language such as Lisp because the language does not enforce sufficient static type restrictions on user programs in order for a compiler to gather all the necessary type information for later reconstruction. Maintaining tags on every object is the only way to ensure dynamic type consistency. Similarly, in the implementation of lazy languages such as Haskell [PJ92], all objects are *boxed* into closures to ensure lazy evaluation semantics. These closures can easily identify themselves and the object they contain *via* their code pointers. Independent type reconstruction does not provide any advantage in this situation.

However, for the class of statically-typed languages that follow applicative-order evaluation,[1]

[1] By applicative-order evaluation, we mean languages that evaluate function arguments before or in parallel with the invocation of the function.

type reconstruction enables substantial representational savings without sacrificing any run-time information. The object representations can be made clean and simple just like in C and Fortran, without compromising type consistency or the ability to use type information at run-time. Of course, we need to ensure that complete type reconstruction is possible for all run-time objects under all circumstances. However, the existing schemes [App89, Gol91, GG92] do not guarantee complete type reconstruction for all run-time objects under all circumstances. In particular, polymorphism and higher-order functions pose significant problems as discussed below.

5.2.5 Polymorphism and Higher-order Functions

Language features such as polymorphism and higher-order functions significantly complicate the problem of making exact type information available in a run-time system with untagged objects. Polymorphic functions are designed to be reusable with various types of data objects, therefore no clue about the type of an object may be associated with the *definition* of such a function. The exact run-time type of a particular application of a polymorphic function is usually an *instantiation* of its static type and must be derived from the *use* of the function at that application site. The run-time system needs to compute such instantiations upon a type reconstruction request.

Similarly, higher-order functions take function closures as arguments and produce closures as results. These function closures may encapsulate hidden objects that are bound to the free identifiers of the function. Unfortunately, even an exact instantiation of the type of a function closure may not reflect the types of the objects captured within its environment. Therefore, the types of objects hidden within higher-order function closures may be impossible to reconstruct. We will examine some of these problems and their possible solutions in Chapter 6.

5.2.6 Type Inference *vs.* Type Declaration

Type inference is a convenient mechanism that frees the user from the task of declaring every identifier in the program with an appropriate type. Most modern programming languages such as Standard ML, Haskell, and Id use a systematic type inference system [Mil78]. Even languages favoring type declaration such as Pascal and C perform some *ad hoc*type inference in order to support automatic type coercions.

Type reconstruction may be thought of as run-time type inference on the dynamic state of the computation, although, a large amount of that information is pre-computed statically within the compiler. The use of type reconstruction at run-time is orthogonal to whether the compiler uses type inference or type declarations in order to collect the necessary static type information. Providing the type information within the program in the form of type declarations does not reduce the complexity of making that information available at run-time. The compiler still has the task of saving all the necessary information in the appropriate form and making sure that complete type reconstruction is possible for all objects at run-time due to the problems discussed above.

5.3 Our Approach

Id is a strongly and statically-typed language. Furthermore, it supports a polymorphic type inference system and uses an untagged run-time system. Our goal is to use run-time type reconstruction in order to determine the exact type of all objects within the Id run-time system.

As mentioned earlier, the existing schemes [App89, Gol91, GG92] are unable to reconstruct the types of some objects. We would like to fix this situation so that the *exact* type of *all* run-time objects may be reconstructed automatically.

Our proposed scheme lies somewhere in-between the two extremes of complete run-time tagging of objects (*à la* Lisp, Standard ML) and carrying no type information at all (*à la* C) without compromising the goal of complete run-time type reconstructibility. We do not tag every run-time object, although a small amount of explicit type information may have to be carried within some higher-order, polymorphic functions in order to allow complete run-time type reconstruction. We analyze the user program at compile-time to detect such cases and insert the additional type information automatically. Essentially, our scheme can be viewed as compiler-directed explicit tagging for such run-time objects. We also provide a type reconstruction algorithm and prove its correctness. The success of our scheme depends on the fact that the explicit type information needs to be inserted in very few cases that essentially plug the informational holes in the previous schemes and that it can be set up by the compiler automatically with little run-time support and overhead.

The main contribution of this work is that we guarantee complete type reconstruction. As we will see in Chapter 7, our current system slightly restricts the acceptable set of type-correct programs in order to provide this guarantee. On the other hand, this guarantee opens the way for a universal framework for supporting various language and system applications that need to use exact object type information at run-time. We discuss some of these applications below.

5.4 Applications of Complete Run-time Type Reconstruction

5.4.1 Polymorphic Source Debugging

A Source debugger for a statically-typed, polymorphic language is an ideal application for run-time type reconstruction. In a debugger, it may be necessary to display the values of any or all of the variables associated with a given procedure activation. Without any help from the run-time system, the static type signatures of polymorphic objects are usually insufficient to traverse and display their full contents. For example, the `append` function on lists has the polymorphic static type $\forall t_0.\ (list\ t_0) \rightarrow (list\ t_0) \rightarrow (list\ t_0)$. The function may be used in various contexts to append various kinds of lists. In each case, we need to reconstruct the full run-time type of its arguments in order to display their contents appropriately to the user.

Another interesting property of source debugging is that type reconstruction is required only for those objects (or function activation frames) that are requested by the user for displaying. The entire state of the machine need not be reconstructed at once. Moreover, debugging does not impose any serious performance constraints for type reconstruction. Users are generally willing to tolerate a reasonable cost for displaying an object which would now also include the cost of reconstructing its type.

5.4.2 Tagless Garbage Collection

Type reconstruction may also be used within a run-time system in order to perform garbage collection without maintaining any type information on the heap objects themselves. Abstractly, a garbage collector performs two functions: it distinguishes live objects from those that are garbage (*live-object detection*), and it reclaims the storage allocated to objects that are garbage (*dead-object reclamation*). For live-object detection, the garbage collector must be able to distinguish scalar objects from heap-allocated objects and determine their sizes (*object identification*). The actual type of an object is very useful for this purpose.

Conventional techniques for object identification operate with a very simple memory model and make little or no use of language and compiler-specific information. Pointers may be tagged using one bit to distinguish them from scalars values and objects may be provided with header tags or may be allocated in separate areas of memory to keep track of their size. The reader is referred to a recent such techniques in [Wil92].

Unlike source debugging, garbage collection does not require complete source type information per say, but additional type information may be helpful in optimizing the marking of live objects. For instance, it may be possible to entirely skip the traversal of large arrays while searching for embedded pointers to heap objects, if the exact run-time type of their elements turns out to be a scalar. Clever compilers and run-time systems that tag every object [App90] may sometimes be able to encode such information within the header of the array if its type is statically known to be a scalar, but this is not possible with polymorphic array constructors such as the `make_vector` function of Example 2.1 which could be used in both scalar and structured array computations.

An alternative solution for object identification is to use complete run-time type reconstruction. This technique enables garbage collection to be performed in an untagged run-time system, saving valuable application time and space spent in continuous tag maintenance. Complete type information also paves the way to type-based optimizations in marking flat data-structures as discussed above. But, one has to weigh these advantages against the cost of performing type reconstruction whenever garbage collection is requested.

As an example, a simple "mark-and-sweep" tagless garbage collector would work as follows. When garbage collection is initiated, the first step would be to reconstruct the types of the root set of heap objects that are either stored in global variables or pointed at from within the function activation frames. The reconstructed type information would then be used to guide the garbage collector in identifying and traversing the reachable heap objects and marking them as live. Finally, unmarked objects would be reclaimed as garbage. We describe such a scheme in Chapter 8.

5.4.3 Object-based I/O

Another application that may benefit from run-time type information is I/O. Most programming languages offer either stream-based or continuation-passing I/O primitives for a few basic datatypes that may used to build more complex read/write functions explicitly (*e.g.*, C, Pascal, Haskell). Typically, I/O formats and styles for complex objects are directly controlled by the user. Polymorphic objects are handled using explicitly parameterized I/O routines. With run-time availability of type information, I/O handling for complex (even polymorphic) objects can be made automatic. The structure of an object may be directly determined from its type. For fixed sized objects, the size of the object may also be ascertained from its type. For dynamically sized arrays, the size information may be kept within the object itself. Given this information, an entire complex object may be read or written easily using its type to select and guide the output format.

The run-time systems of dynamically-typed polymorphic languages such as Lisp or Smalltalk usually offer such I/O capability automatically for each user-defined data-structure within the program. This is possible because all objects in such languages carry type-tags which may be used to guide the generic I/O functions according to the structure of that object. With type reconstruction, this capability may also be provided in a statically-typed languages with an untagged object model. Moreover, just like tagless garbage collection, it may also be possible to generate object-based I/O routines that are specialized to a given object type and hence are more efficient than generic I/O routines that interpret the reconstructed type at run-time.

Another possible use of complete run-time type reconstruction and object-based I/O is in periodic check-pointing of the entire machine state for long-running programs. Complete type reconstruction would enable traversal and recording of all the dynamic data-structures participating in the computation including the activation stack, the global environment, and all the accessible objects residing on the heap.

5.5 Outline

In the rest of Part II we study the problem of complete run-time type reconstruction for Id programs in detail and describe some of its applications implemented within the Id run-time system. In Chapter 6, we intuitively analyze the problem of polymorphic type reconstruction by means of examples, describe the compiler and run-time system support required and outline a reconstruction algorithm. Chapter 7 formalizes these ideas in the context of the Kernel Id intermediate language, presents a complete reconstruction algorithm, and proves its correctness. Finally, in Chapter 8 we present tagless garbage collection as an application of complete type reconstruction and compare its performance with a conservative garbage collector and a compiler-directed explicit allocation/deallocation scheme.

Chapter 6

Compiler-directed Polymorphic Type Reconstruction

In this chapter, we informally present the problem of complete run-time type reconstruction for higher-order, polymorphic languages such as Id and discuss some of its solutions. In Section 6.1, we briefly describe the problem *via* examples and discuss why the existing approaches are insufficient to guarantee complete run-time type reconstruction. In Section 6.2, we provide the basic framework for doing complete type reconstruction, characterizing the analysis required at compile-time and the reconstruction strategy to be followed at run-time. Next, in Section 6.3 we present a compilation scheme that identifies and inserts the necessary type information within the user program to guarantee complete type reconstruction at run-time. Subsequently, Section 6.4 walks through a reconstruction example. In Section 6.5, we show a series of compiler optimizations and variations on our compilation scheme that may further reduce the book-keeping overhead of the current scheme. Finally, in Section 6.6 we point to two implementations of our type reconstruction strategy.

6.1 Type Reconstruction Problem

The problem of type reconstruction for Id can be described as follows. At some point during the execution of a program, we wish to take a snapshot of the state of the machine and determine the type of every object accessible within the computation. We assume that the program is typed statically and that the run-time environment does not maintain *any* type information implicitly. In particular, Id run-time objects do not carry any type-tags.

Clearly, only polymorphic objects and functions pose some challenge; complete type information can be obtained at compile-time for monomorphic objects. Also note that the exact nature of the desired information depends on the application that uses it. For example, a source debugger may wish to inspect any particular object from the current run-time state of the machine whereas a garbage collector only needs to traverse those that are still in use. Also, most garbage collectors only need to differentiate between scalars and pointers to structures while a source debugger needs exact type information in order to display the object properly. In general, we would like to devise a flexible strategy that can be optimized according to the level of information desired.

6.1.1 Basic Type Reconstruction Scheme

Usually, the compile-time type of an object is a good starting point for the reconstruction of its run-time type. In case of polymorphic functions, the types of the objects contained within the function body would depend on the types of the arguments that it receives at a given application site. Appel [App89] first noted that if the exact types of the arguments of a polymorphic function were known at run-time, then its entire body could be instantiated appropriately using its compile-time typing. The exact types of the arguments present at an application site may, in turn, be determined by reconstructing the type of the parent's body containing that application site and so on.

Goldberg [Gol91] made the above ideas more concrete in the context of tagless garbage collection for strongly-typed, sequential languages. Although his scheme applied specialized garbage collection routines to heap objects directly without explicitly reconstructing their types, the basic mechanism of type reconstruction remains the same and may be described as follows in the context of parallel program execution:

Compile-time support:

1. The program is type-checked completely.
2. For each user-defined function within the program, the types of all its arguments and the types of its local and free variables are recorded in a *type-map*. This type-map serves as a static template for the function's run-time activation frame.
3. For each function application site, the full static type instantiation of the function being applied is also recorded within the type-map of the enclosing function definition.

Program invocation and execution:

1. The top-level expression is type-checked and the types of its command-line arguments are recorded.
2. The top-level expression is now executed, expanding the run-time state of the machine into a tree of *activation frames* (a stack of activation frames in a sequential language). Each function application evaluates in the context of its own activation frame which stores its actual arguments and saves the values of temporary local computations.
3. The machine may be halted at any point during execution and type reconstruction may be requested for a particular frame present within the current dynamic activation tree (the activation stack in a sequential language).

Run-time type reconstruction:

1. First, the function corresponding to the current activation frame is identified and its static type-map is obtained.
2. If the current function is not polymorphic then no type reconstruction is required. Otherwise, its parent activation frame and application site are identified using the return address information in the current frame.
3. If the parent activation frame is the root of the dynamic activation tree then the exact types of the arguments supplied to the current function are already known. Otherwise, the process of type reconstruction is repeated for the parent frame by going back to Step 1.

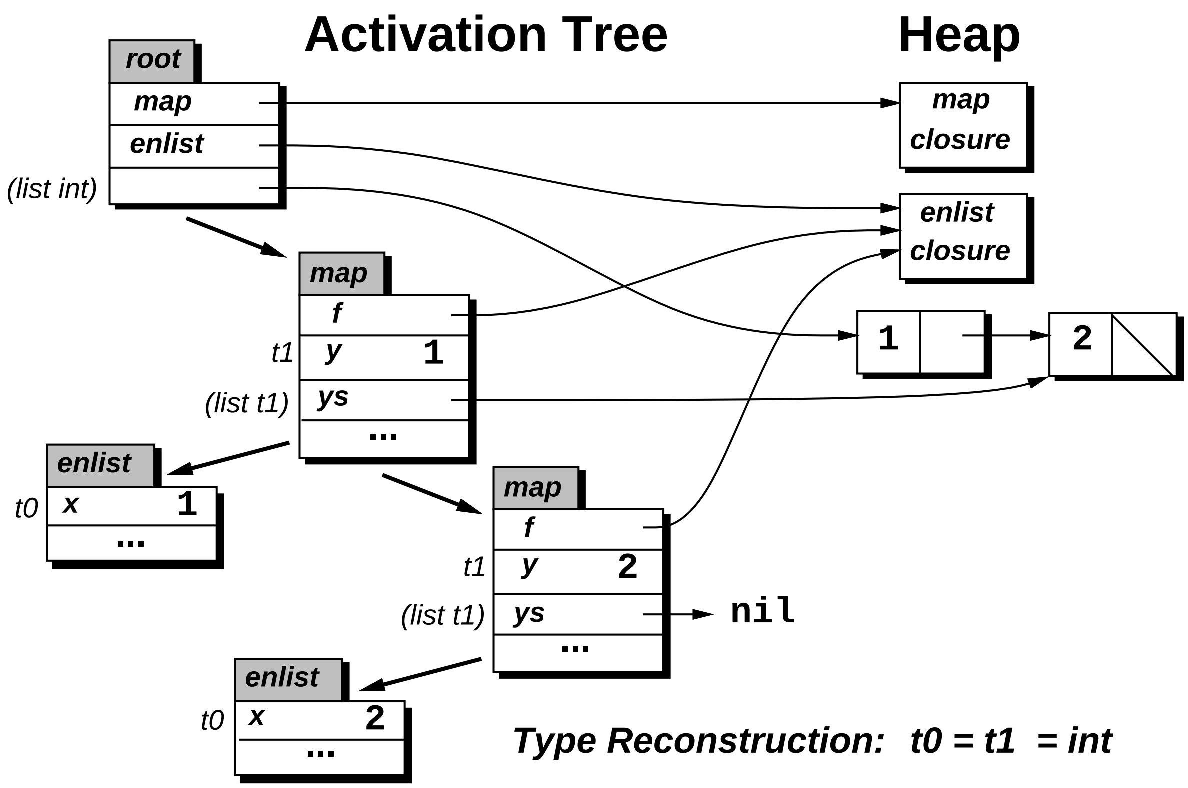

Figure 6.1: The Run-time State of Computation in Example 6.1.

4. Given the exact types of the arguments of a function, its static type-map is fully instantiated by matching the actual types of the arguments to their static types. This reconstructs the current activation frame and also provides the exact types of the arguments present at any application sites within the body of that function.

As shown above, the reconstruction process may continue possibly up to the root of the activation tree where the run-time types of the user-supplied arguments are available. At that point, all polymorphic functions in the call chain can be correctly instantiated revealing the run-time types of their internal objects. In the context of sequential execution, Goldberg [Gol91] also showed that the entire state of the machine may be reconstructed in one pass by starting from the root frame at the bottom of the activation stack and working towards the most recent frame at the top of the activation stack.

We illustrate the above reconstruction scheme with a small example:[1]

Example 6.1:

```
def enlist x_t0 = x:nil;
def map f nil = nil
 |  map f (y:ys)_(list t1) = (f y_t1):(map f ys);

map enlist (1:2:nil)_(list int);
```

[1] All the examples in this chapter use the Id language syntax [Nik91]. Briefly, functions are introduced with a `def` keyword and allow pattern-matching on their arguments. (:) is the infix *cons* operation.

The function `enlist` has a static type $\forall t_0.t_0 \rightarrow (\mathit{list}\ t_0)$ and `map` has a static type $\forall t_1 t_2.(t_1 \rightarrow t_2) \rightarrow (\mathit{list}\ t_1) \rightarrow (\mathit{list}\ t_2)$. We also show the type instances of some internal identifiers as subscripts. The evaluation of the top-level expression `(map enlist (1:2:nil))` dynamically unfolds into a tree of activation frames as shown in Figure 6.1.

If we wish to examine the `x` argument of `enlist` during one of these calls, then the run-time instantiation of its static type t_0 may be determined by following up the dynamic chain of activation frames into its application site within the `map` function. Here, t_0 may be related to the static type t_1 of the actual argument `y` at that application site. This relates to the type of the second argument $(\mathit{list}\ t_1)$ of `map` which is found to be $(\mathit{list}\ \mathit{int})$ at the root application site. Then, both t_1 and t_0 can be instantiated to int giving the actual type of `x` as desired.

6.1.2 Problems with Closures and Free Variables

Unfortunately, the above scheme is incomplete. Goldberg and Gloger [GG92] noted that sometimes types of objects hidden inside a closure are impossible to reconstruct. Consider the following example:

Example 6.2:

```
def f2 x_t0 y_t1 = y;
g2 = if ... then f2 1_int else f2 "foo"_string;

g2 2;
```

Here, `f2` has a type $\forall t_0 t_1.t_0 \rightarrow t_1 \rightarrow t_1$, and therefore `g2` gets bound to a partially applied function closure with type $\forall t_2.t_2 \rightarrow t_2$ that says nothing about the type of the data hidden inside it. In fact, this type cannot be determined at compile-time because it depends on the value of the predicate (...). Besides, during the evaluation of `(g2 2)` the return address information on the call stack would point to the application site of `g2`, which does not help in determining the contents of that closure either. Thus, we cannot reconstruct the type of the argument `x` within the activation of `f2` because the computation that created its closure is no longer available as part of the dynamic activation tree.

It may appear that this problem arises only when an argument of a function is never used within its body, but the following example adapted from [GG92] shows that this is not the case:[2]

Example 6.3:

```
def f3 x_(list t0) =
  { def h3 z_t1 = if length x_(list t0) == 1
                  then z:nil
                  else z:z:nil;
    in h3 };
g3 = if ...
     then f3 (1:nil)_(list int)
     else f3 (true:nil)_(list bool);

g3 2_int;
```

Here, the type of the function `f3` is $\forall t_0 t_1.(\mathit{list}\ t_0) \rightarrow t_1 \rightarrow (\mathit{list}\ t_1)$, and therefore the type of the computed closure `g3` is $\forall t_2.t_2 \rightarrow (\mathit{list}\ t_2)$. During the evaluation of the application `(g3 2)`,

[2]In Id syntax, a block-expression (bounded by {}) encloses a set of identifier bindings. The result of such a block is the value of the expression following the keyword *in* evaluated within the scope of the bindings.

no information is available in the activation tree whether this closure contains a list of booleans or a list of integers. Goldberg and Gloger argue in [GG92] that since `h3` does not use the elements of its free variable list `x` but only its spine (to compute its length), a garbage collector can ignore these elements and copy just the spine. But this approach creates problems if these structures were shared in many places and is quite unsatisfactory for a source debugger that needs to display the full object.

The problem of not being able to reconstruct the exact type of an object as shown above does not appear all the time. For instance, the type of argument `z` within `h3` in the above example may be reconstructed to the type *int* by traversing up the call stack to its application site `(g3 2)`. In fact, functions like `map` in Example 6.1 never have this problem:

Example 6.4:

$\texttt{g4 = (map}$
$\texttt{enlist)}_{(list\, t_0)\rightarrow(list\,(list\, t_0))}\texttt{;}$
$\texttt{g4 (1:2:nil)}_{(list\, int)}\texttt{;}$

Even though here `map` is partially applied to `enlist` to yield a closure `g4` with type $\forall t_0.(list\ t_0) \rightarrow (list\ (list\ t_0))$, we have not lost any type information. Instantiation of t_0 to *int* at the call site of `g4` yields complete type information about all the internal identifiers of both `map` and `enlist`. The problem with Examples 6.2 and 6.3 is that sometimes the types of closures do not have any connection with the types of objects hidden inside them. In such cases, we are in danger of losing type reconstruction information because the closure creation site may no longer be available on the call stack.

Another interesting point is that polymorphic objects with universally quantified types do not pose this problem. The run-time type of such an object cannot be more specific than its compile-time definition type. For instance, in the following example the variable `x` within the body of `f5` has the universally quantified type $\forall t_0.(list\ t_0)$.

Example 6.5:

```
def f5 y =
  { x = nil;
    def h5 z_{t1} = if length x_{(list t0)} == 1
                    then z:nil
                    else z:z:nil;
   in h5 };
```

Now, there is no question about the contents of the closure formed by `h5` over its free variable `x`. It can never contain an object whose type is more specific than $\forall t_0.(list\ t_0)$. For our purposes, this means that the compile-time type of a polymorphic object provides sufficient information for its run-time type reconstruction.

6.1.3 Discussion

The examples presented above attempt to provide an intuitive understanding of the process of type reconstruction. It appears that for some polymorphic functions we are able to infer type reconstruction information from the parent-child relationships embedded in the activation tree while for others we need additional information at run-time for complete type reconstruction. Now we can characterize the problem of type reconstruction more concretely:

1. First, we need to identify and record all the compile-time type information necessary for type reconstruction. We also need a criterion to identify what additional type information,

if any, needs to be carried at run-time for complete type reconstruction of polymorphic functions (Section 6.2).

2. Next, we need a compilation scheme that transforms the given program into one that generates and propagates the additional type information (Section 6.3).

3. Finally, we need a type reconstruction algorithm that uses the explicit and implicit type information at run-time and reconstructs the exact type of all run-time objects (Section 6.4).

6.2 Type Reconstruction Framework

In this section, we discuss the general framework for run-time type reconstruction. First, we describe the run-time execution model of Id programs. Using this model, we formulate a strategy for reconstructing the complete run-time machine state. Finally, we identify the essential information that needs to be recorded at compile-time and establish a type conservation criterion that guarantees complete run-time type reconstruction.

6.2.1 Run-time Model of Program Execution

Id is a non-strict, implicitly parallel language with an eager evaluation strategy. Below, we summarize the execution model of a Kernel Id program.

A program in Kernel Id consists of an expression query to be evaluated within the scope of a set of top-level value bindings and type declarations. Typically, this evaluation is carried out in several phases as described below:

Compile-time — First, the top-level bindings and type declarations are type-checked giving rise to the *global static environment*. This environment records the exact types of all global identifiers. Subsequently, all top-level value bindings, datatype constructors, and internal function definitions are compiled into independent *code-blocks*.

Link/Load-time — All code-blocks are loaded and linked into the program memory giving rise to the *global dynamic environment*.

Invocation-time — The top-level expression query is type checked in the global static environment and then compiled into a *root code-block*. At this point, exact types for all local and free identifiers used in the query expression are known. The global static and dynamic environments together with the typed root code-block for the query expression constitute the complete *initial state* of the machine.

Run-time — A code-block always executes in the context of an **activation frame** which records the actual arguments bound to its formal parameters, the run-time objects bound to its free identifiers, and the values of all its local identifiers during execution. An activation frame is allocated at the time of a function application and it is deallocated when that function terminates. In a sequential system, an activation frame corresponds to the stack frame of the currently executing function. In the parallel execution model of Id, the run-time stack generalizes to a tree of activation frames as shown in Figure 6.2.

The program starts execution by allocating an activation frame for the root code-block recording its actual arguments and local identifiers. Subsequent function invocations extend the dynamic activation tree with their own activation frames, executing in parallel

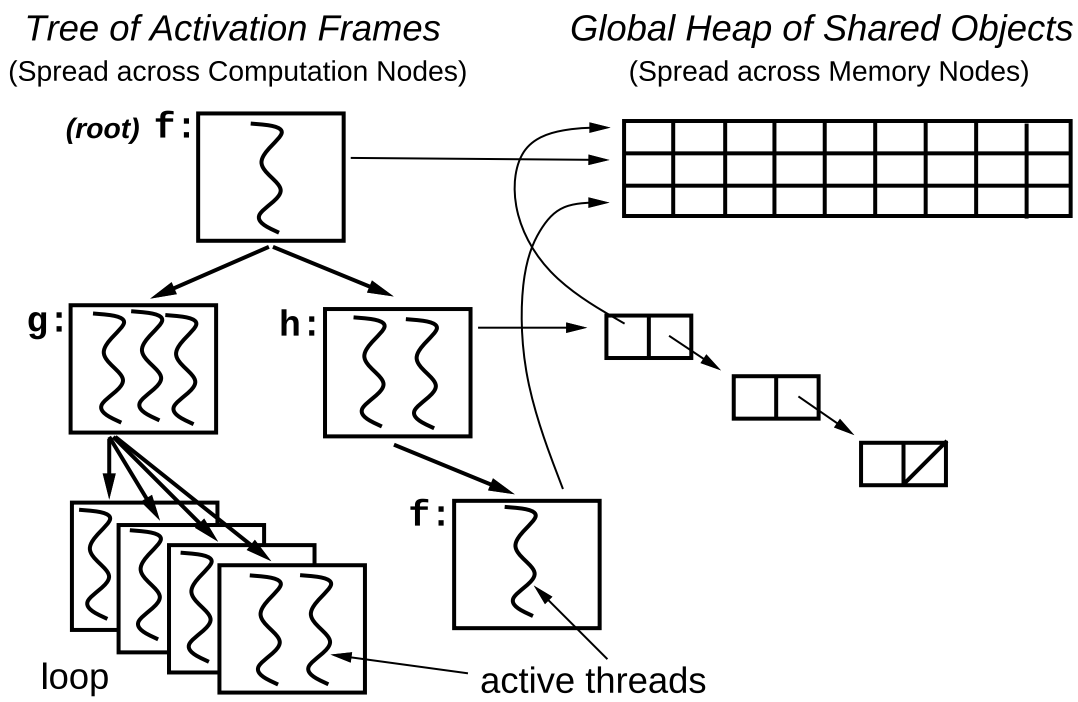

Figure 6.2: The Parallel Execution Model for Id.

with their parent activation. Shared objects are allocated on a separate global heap and are accessible *via* pointers from the activation frame (see Figure 6.2). Thus, at any time during execution, the complete *run-time state* of the machine consists of the global dynamic environment, the tree of active or suspended activation frames, and all the heap objects accessible through the global identifiers or the activation frames. This is the state of the machine we are interested in reconstructing.

6.2.2 Type Reconstructibility

Starting from the initial state of the machine as described above, we can view type reconstructibility as an invariant condition to be maintained at each subsequent evaluation step that modifies the run-time state of the machine. We identify two kinds of state modifications: *intra*-procedural, and *inter*-procedural.

The *intra*-procedural modifications to the state of the machine are due to the computation within a code-block: accessing values of function parameters and free identifiers to compute local values, allocating heap objects, modifying global or heap objects etc. Since our language has a sound type system, type-correct programs are guaranteed not to produce run-time type-errors or to compute values that are type-inconsistent. This implies that any value bound to an identifier in a given code-block must be consistent with the exact type of that identifier, otherwise it could lead to a run-time type-error. This is true even for identifiers bound to mutable objects. In other words, the actual values of mutable identifiers and heap objects could change due to side-effects, but the types of those values would remain the same. Therefore, once the exact types of all identifiers present within a code-block are determined, they serve to identify the exact types of all the values computed and the heap objects allocated within the code-block over its entire life-time.

The *inter*-procedural modifications to the state of the machine take place at a function application or return. A function application introduces a new activation frame that binds a new set of local identifiers and points to the heap objects allocated within the function. We need to ensure that the exact types of these new local identifiers and heap objects are reconstructible on the basis of the existing state of the machine before the function application.

The above discussion suggests that an activation frame is an appropriate unit of type reconstruction. The entire state of the machine may be reconstructed by induction on the structure of the dynamic activation tree. As the base step, the exact types of all objects in the root activation frame are already known at the start of the program. The inductive step is to ensure that at every function application site that expands the dynamic activation tree, the type of every slot in its activation frame can be identified and correctly instantiated. Below, we analyze the compile-time information required to achieve this.

6.2.3 Recording Compile-time Type Information

In Section 6.1.1, we informally introduced the concept of the *type-map* of a function that was used as a static template during its type reconstruction. Below, we make that definition more concrete:

Definition 6.1 (Type-map) *Given a function $f = \lambda x_1 \cdots x_n.E$ with free identifiers $\{z_1 \cdots z_m\} = \mathcal{F}(\lambda x_1 \cdots x_n.E)$ and locally bound identifiers $\{y_1 \cdots y_l\} = \mathcal{B}(E)$, its* **type-map** *denoted by TM_f records the following information:*

1. *The function type, $f : \tau_1 \to \cdots \tau_n \to \tau_{n+1}$.*
2. *The types of all the function parameters $x_1 : \tau_1, \ldots, x_n : \tau_n$.*
3. *The type-schemes of all the free identifiers of the function, $z_1 : \sigma_{z_1}, \ldots, z_m : \sigma_{z_m}$.*
4. *The type-schemes of all the locally bound identifiers $y_1 : \sigma_{y_1}, \ldots, y_l : \sigma_{y_l}$.*
5. *The type-instance of the function identifier g at all application sites $(g\ a_1\ \cdots\ a_k)$ within the function body E. We also record whether an application site has been statically determined to be a full-arity application site.*

A type-map records the static types of all the parameters, the free identifiers, and the local identifiers of a code-block along with some additional type information about its internal call sites. It is essentially a mapping from the frame slots of a code-block's activation frame to their static types. The type-map TM_f is parameterized by the set of all its free type-variables $\mathcal{F}(TM_f)$. This set exactly captures the missing information in the static type environment of a function that needs to be instantiated at run-time.

We generate static type-maps for all code-blocks within the program at compile-time. These templates are then linked together with the compiled object code and may be accessed at run-time using the name of the code-block. As an example, Figure 6.3 shows the Kernel Id translation and the type-map for the `map` function of Example 6.1.

6.2.4 The Principle of Type Conservation

Consider a first-order application site for a function that does not have any free identifiers. We can reconstruct the types of all objects in its activation frame using the basic type reconstruction scheme described in Section 6.1.1. We assume that the name of the callee function can be identified from its current activation frame which also identifies its static type-map. The return

```
def map f l =
 { p = nil? l;
   l1 = if p then nil
        else
         { x = hd l;
           xs = tl l;
           y = f x;
           ys = map f xs;
           l2 = y : ys;
          in l2};
  in l1};
```

Typemap *t0,t1*

map	*(t0 -> t1) -> (list t0) -> (list t1)*	
f	*(t0 -> t1)*	*Arguments*
l	*(list t0)*	
p	*bool*	*Local Frame Slots*
l1	*(list t1)*	
x	*t0*	
xs	*(list t0)*	
y	*t1*	
ys	*(list t1)*	
l2	*(list t1)*	
f	*(t0 -> t1)*	*Internal Call Sites*
map	*(t0 -> t1) -> (list t0) -> (list t1)*	

Figure 6.3: Kernel Id definition and the Type-map of `map` function.

address information stored within the frame identifies the caller's activation frame and the exact application site within the caller's body that gave rise to the call. Assuming that the caller's frame has been reconstructed recursively, the exact type instantiation of the callee function recorded within the type-map of the caller (Item 5 in Definition 6.1) provides the exact types of all the arguments passed to the callee at this application site. Now, the callee function's type-map may be instantiated by matching the types of the actual arguments with the types of the parameters recorded in the callee's type-map.

Unfortunately, not all application sites are first-order, since our language allows higher-order functions and partial applications (currying). As shown in Figure 6.4, partial applications create function closures that simply record the supplied argument in a closure data-structure instead of creating a new activation frame right away. The type of such closures may not provide sufficient information regarding the type of the arguments captured within the closure (*e.g.*, closure `g2` of Example 6.2). Some functions refer to free identifiers that must also be recorded in a closure at the point of their definition[3] (*e.g.*, function `h3` of Example 6.3). The types of such free identifiers may not be reflected in the overall type of the function closure either.

In a higher-order language such as Id, function closures are first-class objects, *i.e.*, they may be stored into heap data-structures, passed as arguments to other functions, and returned as values from the function that created them. Therefore, the function definition site or the partial application site that creates a closure is not guaranteed to be accessible when that closure is used in further computation. As shown in Figure 6.4, such application sites are termed as

[3]Lambda-lifting transformation [Joh85] may be used to lift nested functions with free identifiers into top-level *super-combinators* that refer to only top-level identifiers. But, this transformation restricts the type polymorphism of free identifiers and does nothing to change a higher-order program into a first-order program. Therefore, we choose to deal with the problem of free identifiers directly.

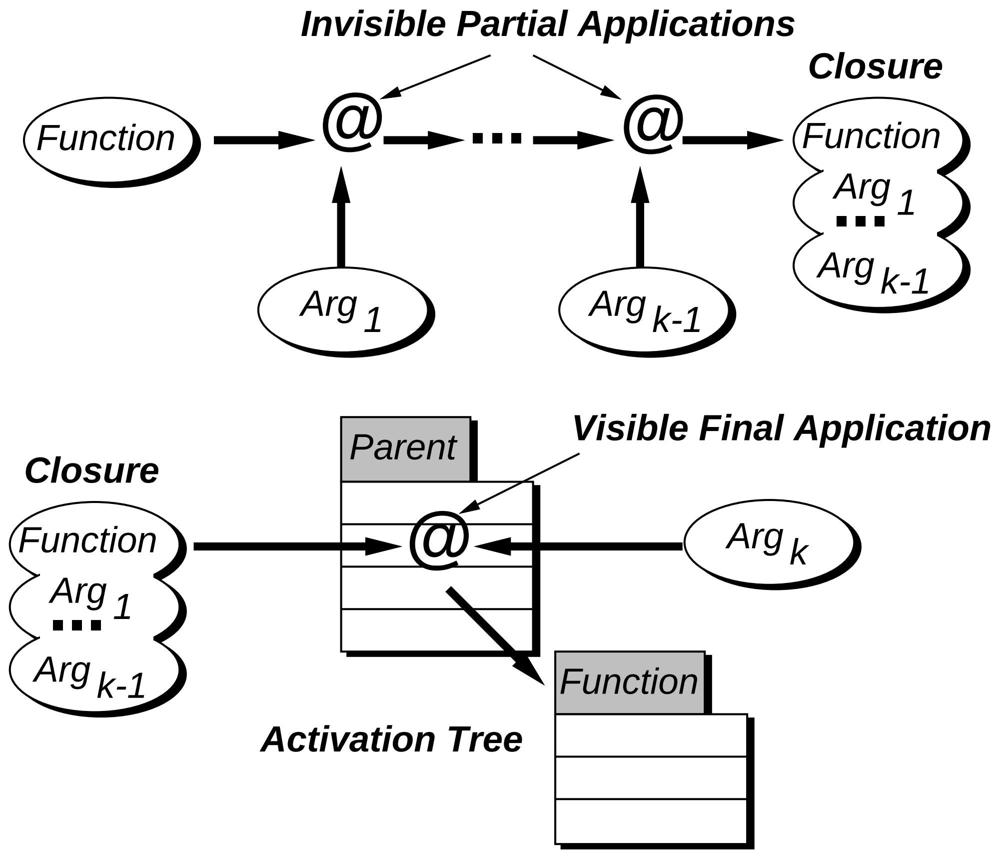

Figure 6.4: Visible and Invisible Application Sites.

invisible. A function closure expands into an activation frame only when all its arguments have been accumulated. This final application site of the closure is termed as the *visible* application site because its position may be determined by examining the return address stored within the expanded activation frame.

The type reconstruction scheme outlined above for first-order function applications would work with higher-order function closures only if the closure type instantiation recorded at the final application site has sufficient type information to instantiate the types of all the free identifiers and previous arguments accumulated within the closure. Such a function is called *type-conserving*. This is a static property of a function's type signature and is characterized in the definition given below. On the other hand, if a function does not satisfy the above property, then some type information may be lost at its definition site or its invisible partial application sites. We also identify such information in the following definition for each of the invisible application sites:

Definition 6.2 (Type Conservation) *Given a function f with arity k, type-map TM_f, and type-scheme $\forall\alpha_1 \ldots \alpha_n.\tau_1 \to \cdots \tau_k \to \tau_{k+1}$,*

1. *The type-variables $\mathcal{F}(TM_f) \backslash \mathcal{F}(\tau_1 \to \cdots \tau_k \to \tau_{k+1})$ are defined as* **not being conserved** *at the function definition site.*

2. *The type-variables $\mathcal{F}(\tau_i) \setminus \mathcal{F}(\tau_{i+1} \to \cdots \tau_k \to \tau_{k+1})$ $(1 \leq i < k)$ are defined as* **not being conserved** *at its i-th application site.*

3. *The type-variables $\mathcal{F}(\tau_k \to \tau_{k+1})$ are defined as* **being conserved** *at the final (k-th) application site.*

4. *The function f is said to be* **type-conserving** *if all the type-variables in its type-map are being conserved, i.e., $\mathcal{F}(TM_f) = \mathcal{F}(\tau_k \to \tau_{k+1})$.*

Informally, a type-conserving function can correctly instantiate its entire type-map with just the run-time type of its final application closure. It is easy to check that `map` and `enlist` from Example 6.1 are type-conserving, while `f2` from Example 6.2, and `f3` and `h3` from Example 6.3 are not, which is why we were losing type information in those cases.

Definition 6.2 may be used by a compiler to detect functions that are not type-conserving. Furthermore, the definition shows exactly how much type information is lost at each application site. The next question is what type reconstruction strategy should be devised for such functions? Our scheme is to make every function closure *self-sufficient*, which means that a closure for a non-type-conserving function must carry exact run-time encodings of its non-conserved types. We describe our compilation scheme in the next section.

6.3 Compiler Support for Type Reconstruction

In this section, we informally describe a compilation scheme that analyzes every function in the program and transforms it to generate and propagate exact run-time type instantiations of its non-conserved type-variables where necessary. These encoded *type-hints* are inserted at the partial application sites that otherwise do not preserve this information and are deposited into the function's activation frame at the time of its final application. These type-hints may then be used to reconstruct the exact type instantiations of the non-conserved type-variables for the current activation frame of the function.

It is interesting to note that the propagation of type information from closure creation sites to their final application sites for non-type-conserving functions may be formulated as an *overloading resolution* problem which may then be handled using well-known techniques in the literature [Gup90, PJW92, WB89]. These techniques systematically translate overloading into parametric polymorphism by replacing unresolved instances of overloaded identifiers in a function with additional parameters that are supplied at its application site. In our scheme, these parameters are the explicit *type-hints* that are used by the type reconstruction algorithm.

Below, we intuitively describe our compilation strategy by means of examples. We also provide a simplified but self-contained description of overloading resolution and translation mechanism as applied to type reconstruction. The full details of this transformation and the subsequent reconstruction process appear in Chapter 7.

6.3.1 Detecting Violations of Type Conservation

The first step in our compilation process is to identify the functions in the program that may require additional type-hints for the non-conserved type-variables in their type-map. This is straightforward given the test for type conservation in Definition 6.2. First, we type-check each function f in the program and generate its type-map TM_f according to Definition 6.1. Then, using Definition 6.2 we determine which type-variables in its type-map, if any, are not

```
def h3 z =
  { y0 = length x;
    y1 = (==) y0 1;
    y2 = if y1 then
         { y3 = z:nil;
          in y3 }
         else
         { y4 = z:nil;
           y5 = z:y4;
          in y5 };
   in y2};
```

Typemap

	t0,t1	Type Parameters
h3	t1 -> (list t1)	Fn. Signature
z	t1	Fn. Arguments
x length	(list t0) ∀t2. (list t2) -> int	Free Identifiers
y0 y1 y2 y3 y4 y5	int bool (list t1) (list t1) (list t1) (list t1)	Local Frame Slots
length	(list t0) -> int	Internal Call Sites

Figure 6.5: The Kernel Id definition and type-map of function `h3` from Example 6.3.

being conserved. For example, the type-map for function `h3` from Example 6.3 is shown in Figure 6.5. Its type signature is $\forall t_1 . t_1 \rightarrow (\mathit{list}\ t_1)$. Comparing these two together we get,

$$\begin{aligned} \mathcal{F}(TM_{\mathtt{h3}}) &= \{t_0, t_1\} \\ \mathcal{F}(t_1 \rightarrow (\mathit{list}\ t_1)) &= \{t_1\} \end{aligned}$$

Therefore, the type-variable t_0 is not being conserved in the function `h3` and it requires a run-time type-hint for proper type reconstruction.

6.3.2 Propagating Non-Conserved Type Information across Functions

In general, additional type-hints may need to be propagated within the body of a function not only to reconstruct its own non-conserved type-variables but to pass them on to other functions within its body that require those type-hints. Also, some of the non-conserved type-variables at these internal application sites may get partially or completely instantiated. We need to record these instantiations so that appropriate type-hints may be generated at those sites.

Both the above problems may be addressed by viewing the reconstruction of the non-conserved type-variables as an overloaded operation *trec?* that must be resolved within the body of the given function. Standard overloading resolution mechanism picks up such unresolved overloaded identifiers and arranges the required information to be passed in as a parameter to the function. Subsequent uses of the function ensure that the additional information can be instantiated from the enclosing environment, thereby propagating the requirement outwards, if necessary. We illustrate this process for the function `h3` of Example 6.3:

Example 6.6:

$\mathtt{def\ f3}_{(\mathit{trec?}\ t_0)}\ \mathtt{x}_{(\mathit{list}\ t_0)}\ \mathtt{=}$

$\quad \{\ \mathtt{def\ h3}_{(\mathit{trec?}\ t_0)}\ \mathtt{z}_{t_1}\ \mathtt{=\ if\ length\ x}_{(\mathit{list}\ t_0)}\ \mathtt{==\ 1}$

```
                    then z:nil
                    else z:z:nil;
    in h3_(trec? t0) };
g3 = if ...
     then f3_(trec? int) (1:nil)_(list int)
     else f3_(trec? bool) (true:nil)_(list bool);
g3 2_int;
```

Here, we have added a *predicate*[4] (*trec?* t_0) as an annotation on the function `h3`. In general, a predicate is added to a function's type signature for every non-conserved type-variable in its type-map at the precise argument position where that information is being lost according to Definition 6.2. Subsequently, the standard overloading resolution mechanism automatically propagates this predicate to the place where `h3` is referenced and to the enclosing lexical function `f3` because it remains uninstantiated (and hence unresolved) in its body. Finally, this predicate propagates to the application sites of `f3` where it is completely instantiated according to the types of the arguments being supplied to `f3` and is considered to be resolved.

Intuitively, the propagation of a predicate associated with a function represents a lack of type information locally which must be supplied from the application site where this predicate is instantiated. Note that the predicate need not provide the full type of the argument or the free identifier of the function that requires such information (*e.g.*, the identifier `x` in Example 6.6). It only identifies the instantiations of the non-conserved type-variables present in that type. This is sufficient to fully instantiate the type stored in the function's type-map corresponding to that identifier. This scheme allows us to share the type instantiations of the non-conserved type-variables across several identifier types that contain that type-variable. Thus, the number of external type instantiations needed by a function is limited by the number of non-conserved type-variables and not by the number of its actual parameters or free identifiers present in its type-map.

Another interesting observation is that predicate instantiations involving polymorphic type-variables are always considered as resolved and are not propagated outwards in the light of the discussion in Section 6.1.2. For instance, `g3` in the above example might have been defined as:

Example 6.7:

```
g3 = if ...
     then f3_(trec? (list t)) (nil:nil)
     else f3_(trec? bool) (true:nil);
```

Here, (*trec?* (*list* t)) is an instantiation of `f3`'s predicate according to its polymorphic argument `(nil:nil)`. Even though this predicate has an uninstantiated type-variable t, it is not propagated any further because it is polymorphic at this point. It follows immediately that there can be no unresolved predicates at the top-level because there are no free type-variables in the top-level type environment by construction.

6.3.3 Program Translation

The final step in our compilation process is to add extra hint parameters to the function definitions that have non-conserved type-variable predicates of the form (*trec?* t). Likewise,

[4] We follow the terminology of [Gup90, WB89] where the usual Hindley/Milner type of a function is extended with *predicates* to model overloaded identifiers. In Haskell [HWe90] these are known as *contexts*. The predicate name *trec?* in our scheme stands for *type-reconstructible?*.

a predicate (*trec?* τ) appearing at a function application site is transformed into a type-hint encoding τ that is passed as an explicit argument at that application site.

It is possible to either add one hint parameter for each non-conserved type-variable or group the hints together in a single *hint-record* from which the individual hints may be fetched. Our current scheme adds one hint parameter per type-variable at the position specified by Definition 6.2. This is because passing a small number of additional parameters is currently cheaper in our system than allocating and fetching from heap data-structures.

The compiler keeps a record of the mapping between the non-conserved type-variables of each function and its additional hint parameters. This mapping, also called the *hint-map*, is shown below:

Definition 6.3 (Hint-map) *Given a function* $f = \lambda x_1 \cdots x_n.E$ *with non-conserved type-variables* $\rho = \{\alpha_1, \ldots, \alpha_m\}$, *its* **hint-map** *is the mapping* $HM_f = \{(\alpha_1 \mapsto y_1), \ldots, (\alpha_m \mapsto y_m)\}$, *where* $y_1, \ldots, y_m$ *are its new additional hint parameters.*

As an example, below we show the hint-map for the function `h3` from Example 6.6:

Type Variable	Hint Parameter
t_0	`h3_hint_1`

The actual type-hints may now be generated using an encoding of the type constructors and their type arguments. The encoding should permit type-hint construction and propagation from within the user program. Although not necessary, we may view the encoding as an Id datatype as shown below:

```
type type_hint = none | tc string (list type_hint);
```

The disjunct `none` encodes polymorphic type-variables that do not require any hint. The disjunct `tc` encodes a type-constructor by its name and a list of encoded type-parameters. The free type-variables of a type-hint τ are encoded using the corresponding additional parameters of the enclosing function definition recorded in its hint-map.

Continuing with Example 6.6 above, the following translation is obtained:

Example 6.8:

```
def f3 f3_hint_1 x =
  { def h3 h3_hint_1 z = if length x == 1
                         then z:nil
                         else z:z:nil;
    in h3 f3_hint_1 };
g3 = if ...
     then f3 (tc "int" nil) (1:nil)
     else f3 (tc "bool" nil) (true:nil);
g3 2;
```

Notice how the hints generated within `g3` propagate into `h3` *via* the hint parameters of `f3` and `h3`. The appropriate hint will now be available in a dynamic activation of `h3` where it may be used along with its type-map to reconstruct the exact run-time type of `x`.

6.4 Run-time Type Reconstruction

Now, we have all the necessary information to reconstruct the entire run-time state of the machine. As discussed earlier in Section 6.2.1, the global dynamic environment and the tree of

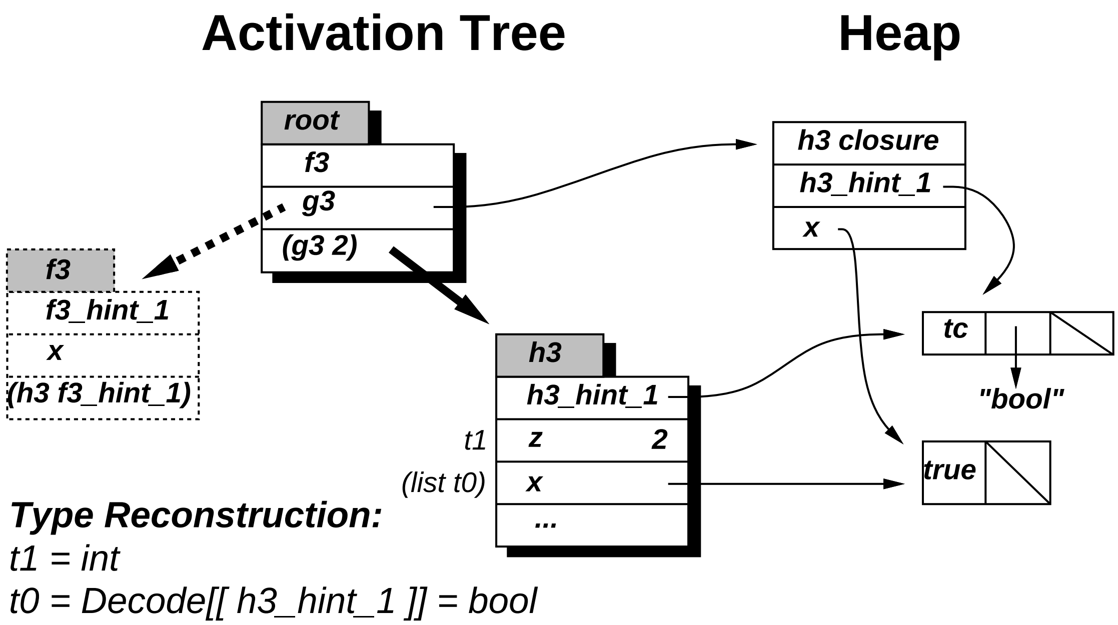

Figure 6.6: The Run-time State of Computation in Example 6.8.

activation frames constitute the root set of the run-time state of the machine. All the relevant heap objects may be accessed through this root set. The types of the global identifiers are already available in the global static environment. Therefore, we only need to reconstruct the types of all the activation frames in order to obtain the types of all the objects in the root set. The type of any accessible heap object may then be reconstructed by examining the fully instantiated type of an appropriate pointer within the root set that leads to the given heap object.

The detailed algorithm for complete type reconstruction of an activation frame will be presented in Chapter 7. Here, we describe a type reconstruction example to illustrate the modifications to the basic scheme presented in Section 6.1.1. These modifications use the type-hints inserted by the compilation scheme of Section 6.3 to account for the type information that is otherwise lost.

6.4.1 A Type Reconstruction Example

Figure 6.6 shows a snapshot of the state of the machine during the execution of translated Example 6.8. Let us suppose that the predicate (...) in the definition of `g3` evaluates to `false` at run-time. The computation of `g3` expands into an activation frame for `f3`, returning a closure for the function `h3` with the appropriate type-hint and the second argument hidden inside. We assume that this computation has terminated and the activation frame for `f3` has been deallocated (shown with dotted lines in Figure 6.6) so that there is no trace of the application site where `g3` was constructed. The evaluation of the application `(g3 2)` unfolds the computation into an activation frame for `h3` as shown in Figure 6.6. Let us also suppose that the program is halted when `h3` has just been invoked. The problem is to reconstruct the types of the objects in `h3`.

The type-map of the function `h3` given in Figure 6.5 shows that it needs the exact type instantiations of the type-variables t_0 and t_1 for proper type reconstruction. From the hint-map given in Section 6.3.3, we know that the additional parameter `h3_hint_1` encodes the exact type instantiation for the type-variable t_0 which is decoded to produce the type *bool*. The type of the free identifier `x` within `h3` may now be reconstructed to be (*list bool*) as given by its type-map. The remaining type-variable t_1 is instantiated to the type *int* as described earlier in Section 6.1.1 by matching the application site type instance recorded in the root type-map with the full type signature of the `h3`. This completely instantiates `h3`'s type-map yielding the exact types of its function parameter `z` and its other local identifiers.

As noted in Section 6.1.2, the type reconstruction schemes described earlier [App89, Gol91, GG92] would fail to reconstruct the type of `x` in the body of `h3`. The reason is that these schemes only use the type information derived from the current stack of activation frames. When higher-order closures such as `g3` are invoked and type reconstructed, the function producing it, `f3`, may not be present on the current stack. Any clues that `f3` might have provided regarding the types of free identifiers of `g3` are therefore not accessible during reconstruction.

6.5 Compiler Optimizations

It might appear that our compilation scheme incurs a lot of run-time overhead due to additional parameters and encoding and decoding of types but our experience has been that realistic programs contain very few (if any) non-type-conserving functions, so the overhead of generating and propagating their type-hints is reasonably small. Although our current performance is adequate, we hope to be able to improve our scheme through several compiler optimizations that are discussed below.

6.5.1 Rearranging the Hint Parameters

Currently, additional type-hint parameters required by a function definition are placed just in front of the regular parameter that would otherwise lose that information according to Definition 6.2. This is not strictly necessary. We can place a hint parameter either *before* or *after* the first regular parameter whose type contains the non-conserved type-variable that is encoded by the hint parameter. This rearrangement does not affect program translation (Section 6.3.3) since the regular parameter and the associated type-hint parameter are still supplied together at the same application site. Of course, the hint parameters corresponding to the non-conserved type-variables in the types of the free identifiers of a function must still be place right up front.

The benefit of such rearrangement is that it may sometimes reduce the propagation overhead of type-hints by removing some extra parameters altogether *via* η-reduction. For example, the following alternate translation for Example 6.6 is also valid (compare with Example 6.8):

Example 6.9:
```
def f3 x =
  { def h3 h3_hint_1 z = if length x == 1
                         then z:nil
                         else z:z:nil;
    in h3 };
g3 = if ...
     then f3 (1:nil) (tc "int" nil)
     else f3 (true:nil) (tc "bool" nil);
```

Here, the parameter `f3_hint_1` of `f3` was pushed after its parameter `x` which made this η-reduction possible.

6.5.2 Arity Analysis

Definition 6.2 conservatively prescribes that the only type-variables that are conserved in a multiple-arity function are those present in its final application type because the function could be curried over its initial arguments. This definition can be specialized to include the types of all the arguments present at an application site, if that site is guaranteed to be accessible through the dynamic activation tree. That is, all arguments at an application site that leads to a full application may be treated as being conserved at that application site. For example:

Example 6.10:
```
def f l1 l2 l3 = (length l1)+(length l2)+(length l3);
g = f (1:2:nil);
...(g (true:nil) ("foo":nil))...
```

Definition 6.2 predicts that the types of lists `l1` and `l2` are not conserved by the definition `f`. But at the final application site for closure `g`, `l2` is also available immediately which implies that its type is conserved at this application site.

In general, at compile-time, it may not be possible to recognize the application of an arbitrary function closure as its final application site. But it is easy to recognize the special case of first-order (or full-arity) application of a function where all its arguments are supplied at once. In such cases, the types of all the actual arguments and the type-variables present in them may be instantiated from its application site, although the function may still require type-hints in order to reconstruct the types of its free identifiers.

In our current scheme, it is not possible to optimize away the type-hints prescribed by Definition 6.2 for a function at its first-order application sites because the function definition may still require type-hint parameters due to higher-order application sites present elsewhere. This is simply a consequence of our choice to provide type-hints by adding extra parameters to a function's definition. Alternatively, we can either generate a specialized first-order version of the function that does not carry any type-hints and use it wherever possible, or choose another mechanism for hint propagation that is transparent to the usual parameter passing conventions. Then, we would be able to tailor the type-hints according to the information available at a particular call site without affecting the function's definition.

6.5.3 Escape Analysis

Together with first-order call site information, if the types of the free identifiers of a function are also known to be reconstructible *via* the currently visible activation tree, then no extra types-hints are necessary at all, even if the function was determined to be non-type-conserving by Definition 6.2. Escape analysis of function closures offers this information. Specifically, if analysis shows that a function closure does not escape from the lexical scope where it was defined, then the correct instantiations of its free identifiers would still be available from the activation frame of this ancestor in the activation tree. In that case, we do not need to set up extra type-hints to reconstruct these instantiations within the given function's activation frame.

It is possible to use the region-based closure typing system described in Part I of this thesis to undertake such escape analysis for internal function closures. We simply need to associate a fresh region variable with each internal function definition that statically tracks the movement of its closure data-structure. Presence of this region variable in the type environment of the enclosing control block, or in the type of the returned value from that block would indicate that the function closure is escaping the scope of its definition.

6.5.4 Tail Calls

Our current scheme does not deal with tail calls where the usual caller–callee relationship is violated. A tail call removes the caller's activation frame from the activation tree and connects the callee to the parent of the caller directly. In such a situation, the application site information for the callee is lost. Consider the following example:

Example 6.11:

```
def f x = 1 + length x;
def g n = if n == 1
          then f (1:2:nil)
          else f (true:nil);

g ...;
```

Without tail calls, the type of `x` in an activation of `f` can be determined by locating its call site within the `then` or the `else` branch of the conditional inside `g`. But, if these applications were compiled as tail calls, then the `f`'s activation will get directly connected to the top-level and the call site information will be lost.

It is easy to extend our scheme to deal with this situation. We simply modify Definition 6.2 to reflect the fact that no call site information is available for `f` and therefore explicit type-hints may be needed for all of its free type-variables. This leads to the following translation:

Example 6.12:

```
def f f_hint_1 x = 1 + length x;
def g n = if n == 1
          then f (tc "int" nil) (1:2:nil)
          else f (tc "bool" nil) (true:nil);

g ...;
```

Now, all the type information is available from within the activation of `f`. Of course, this scheme is not optimal because it ignores the call site information even when it is available using regular calling conventions. In order to incorporate that flexibility, we need to generate several application site specific versions of the function definition as discussed earlier.

6.5.5 Type Specialization

Our current scheme generates and interprets encoded type information in order to reconstruct the types of all local and free identifiers of a function. We do not take any position on what to do with these types. This strategy is adequate and desirable for a source debugger because it may wish to manipulate an object in many different ways. Once the type of the object is reconstructed, it can be interpreted to traverse and manipulate the object in any desired way.

It is possible to apply the principle of type conservation (Definition 6.2) and the program analysis and translation strategy (Section 6.3) in any specific context to allow complete analysis of run-time objects in that context. For instance, in order to display objects in the Id debugger, we could compile a parameterized display routine for every datatype occurring in the program.

Run-time type reconstruction would be used to compose these display routines appropriately and then display the given object directly by passing it to its display routine without any type interpretation.

Similarly, it is possible to generate specialized garbage collection routine(s) for every function instead of its type-map, parameterized by GC-routines that correspond to the free type-variables in its type-map. Then, we can generate and propagate closures of GC-routines instead of type-hints as described in Section 6.3. These parameter routines would be picked up automatically by the GC-routine(s) of the function from its activation frame at the time of garbage collection. This scheme would operate in the same way as the tagless garbage collection mechanism proposed by Goldberg [Gol91] where function-specific and site-specific garbage collection routines are generated that understand the structure and the liveness properties of the local identifiers of a function. Moreover, no additional hash-tables would be necessary in order to keep track of partially traversed polymorphic shared objects as shown in [GG92] because complete type reconstruction ensures that the entire traversal of a shared object can be completed the very first time it is encountered.

6.6 Implementation Status

The type reconstruction scheme described in this chapter has been implemented in two different applications within the Id programming environment. We briefly discuss these implementations below.

6.6.1 Type Reconstruction in a Polymorphic Source Debugger

The need to solve the problem of type reconstruction initially arose while attempting to display polymorphic object within a source-level debugger for Id. A preliminary version of the type reconstruction scheme described in this chapter was implemented during the fall of 1992 in the context of the Id source debugger [Car93] for the Monsoon dataflow architecture and was reported in [AC93].

The Id compiler [Tra86] was modified to perform the type analysis and hint generation for every function within the user program as shown in Section 6.3. A simple Id datatype encoding was used for type-hints as shown in Section 6.3.3. The compiler also generated the type-map and the hint-map for every function. In order to reduce the book-keeping within the debugger, the types of temporary, internal identifiers were dropped from the type-map of a function; only source-level, user-defined identifiers were kept together with their position in the function's activation frame.

The Id debugger [Car93] was written in Lisp and executed on the host processor in the front. It allowed a user to stop the Id program executing on the Monsoon processor in the back when certain pre-specified *events* were triggered. The user could then traverse the current tree of activation frames within the Monsoon memory and request function arguments and local identifiers to be displayed along with any heap objects that they pointed to. Objects within the Id run-time system did not carry any type-tags, therefore, complete type reconstruction was needed in order to decipher the run-time object structure. The debugger reconstructed the object types one frame at a time using the run-time type-hints and the type-map and the hint-map information provided by the compiler. These types were then interpreted to traverse and display the contents of the requested identifiers properly. Objects hidden inside higher-order function closures were not displayed, although such objects could be displayed once the closure was applied and gave rise to an activation frame.

The entire Id programming environment called "Id-World" containing an editor-based incremental Id compiler, a simulator for the Monsoon architecture, and the Id source debugger with complete polymorphic run-time type reconstruction was successfully demonstrated during the ACM Conference on Functional Programming Languages and Computer Architecture held in Copenhagen, Denmark, in June 1993.

In [AC93], we presented a preliminary compilation and type reconstruction scheme which omitted some of the formal details. The complete compilation scheme and the type reconstruction algorithm now appears in Chapter 7 along with a proof of its correctness.

6.6.2 Type Reconstruction for Tagless Garbage Collection

During the fall of 1993, full support for run-time type reconstruction was integrated into the Id compiler for the *T multi-threaded architecture and its run-time system [CCF+93] for the purpose of performing tagless garbage collection. Naturally, this required complete type reconstruction for every slot of every function activation frame and all the heap objects reachable from them including higher-order function closures.

We conducted a feasibility study involving the design and implementation of a simple "mark-and-sweep" garbage collector for the *T architecture based on the run-time type reconstruction mechanism. We compared the performance of this scheme against a conservative garbage collector and a compiler-directed explicit allocation/deallocation scheme, all implemented within the same framework. The results of this study were first reported in [AFH94] and are presented here in Chapter 8. The study showed that tagless garbage collection based on type reconstruction was not only feasible but also beneficial for scientific programs with large scalar arrays. The study also indicated that the type reconstruction cost was a small fraction of the overall garbage collection cost. Complete details appear in Chapter 8.

Chapter 7

Formal Framework for Run-time Type Reconstruction

In this chapter, we formalize the reconstruction strategy outlined in the last chapter. Section 7.1 presents the complete grammar for our intermediate language Kernel Id. In Section 7.2, we describe a compilation scheme that analyzes the source program to identify the additional type information necessary for complete type reconstruction and then transforms the program to propagate this information at run-time. Section 7.3 presents the run-time type reconstruction algorithm and discusses its complexity. Finally, in Section 7.4 we show the correctness of our algorithm.

7.1 The Kernel Id Intermediate Language

Our description of type reconstruction is based on the Kernel Id intermediate language Kernel Id as shown in Figure 7.1. This language supports a rich set of datatypes including typical scalar basetypes, general algebraic (sum-of-products) datatypes, n-dimensional arrays, and curried functions. Records and tuples are a special case of algebraic datatypes with a single product disjunct. We also assume a rich set of primitive functions for basetypes and array construction/selection/modification, as well as standard predefined algebraic datatypes such as *list* and *bool*.

Kernel Id allows multi-arity function definitions and general algebraic type declarations. Every sub-expression in this language is given an explicit name that permits accurate representation of data-sharing. In particular, we assume that every λ-expression has an identifier name associated with it, *i.e.*, λ-expressions are only allowed to occur on the right hand side of a binding. Simply nested `let`-bindings are generalized to a recursive `letrec`-style *block* of bindings. Similarly, a 2-way conditional operator (`if...then...else...`) is generalized to an m-way `Case` dispatch operator. The semantics of this language has been given directly in terms of graph rewriting rules as shown in [AA91, AA94]. Although, we will use the operational machinery described in Chapter 3 while showing the correctness of our type reconstruction algorithm.

Kernel Id is a more realistic abstraction of actual intermediate form used in the Id compiler [AA91, Tra86] than the tiny expression language used in Chapter 3. The Id source language supports special syntactic constructs such as list and array comprehensions, complex pattern matching, and nested function and type declarations [Nik91]. During compilation, the Id source program is translated into a Kernel Id program using standard front-end analy-

Expressions

c	$\in$	Constant
$f, x, y, z \ldots$	$\in$	Identifier
SE	$\in$	Simple Expression
E	$\in$	Expression
PF^n	$\in$	Primitive Fn. with n arguments
$\texttt{Case}^m\_T$	$\in$	m-way Case Dispatch for type T
$C_m^{k_m}$	$\in$	m-th Constructor Identifier with k_m arguments
Constant	::=	$\mathit{Integer} \mid \mathit{Float}$
SE	::=	Identifier | Constant
E	::=	$SE \mid PF^n\ (SE_1, \ldots, SE_n) \mid \texttt{Case}^m\_T\ SE\ (E_1 \cdots E_m)$
		$\mid \lambda x_1 \cdots x_n.\ E \mid SE_1\ SE_2 \mid$ Block
Block	::=	{ [Binding;]* `in` SE }
Binding	::=	Identifier $= E$
Declaration	::=	Binding | Type-Decl
Type-Decl	::=	`type` $T\ \alpha_1 \cdots \alpha_n = \quad C_1\ \tau_{11} \cdots \tau_{1k_1}$
		$\mid\ \cdots$
		$\mid\ C_m\ \tau_{m1} \cdots \tau_{mk_m}$
Program	::=	[Declaration;]* E

Figure 7.1: The Kernel Id Intermediate Language.

ses and transformations such as comprehension-desugaring, scope-analysis, type-checking, and pattern-matching compilation [AA91, Gup90, Tra86]. These transformations result in a Kernel Id program where every sub-expression has a unique name and a well-defined Hindley/Milner type, so that all internal type declarations can be lifted to the top-level. Although, we use source Id syntax in our examples, their correspondence to a Kernel Id program should be easy to follow.

7.2 Compiler Support for Type Reconstruction

7.2.1 A Type System for Computing Type-hints

Figure 7.2 shows a systematic way of performing type-hint analysis and propagation discussed informally in Chapter 6 within the context of the Kernel Id intermediate language. We have modified the usual Hindley/Milner typing rules [Mil78] to compute and propagate additional type-hint information. In this system, the conventional Hindley/Milner type of a function closure $(\tau_1 \rightarrow \tau_2)$ is prefixed with the set of type-variables ρ that are not conserved in its immediately previous partial application.[1]

Definition 6.2 identifies the exact set of non-conserved type-variables at each argument position of a multi-arity, user-defined function. Type-conserving positions are assigned the empty set ϕ. Each type-variable $t \in \rho$ may be taken to represent the overloading predicate $(\mathit{trec?}\ t)$ as shown in Section 6.3.2. Type schemes σ generalize and instantiate such augmented

[1] Although, ρ is defined here to be a *set*, the ordering of the type-variables within the set would become important when we translate their type instantiations into actual type-hints parameters.

TYPES

$$
\begin{array}{lcl}
\alpha, \beta & \in & \text{Type-Variable} \\
T^n & \in & \text{Type-Constructor with } n \text{ type arguments} \\
\tau & \in & \text{Type} \\
\rho & \in & \text{Type-hint Set} = \text{POWER-SET(Type)} \\
\sigma & \in & \text{Type Scheme} \\
TE & \in & \text{Type Environment} = \text{Identifier} \rightarrow \text{Type Scheme}
\end{array}
$$

$$
\begin{array}{lcl}
\tau & ::= & \alpha \mid int \mid float \mid (\mathtt{n}d\_array\ \tau) \\
 & & \mid (T^n\ \tau_1 \cdots \tau_n) \mid \tau_1 \rightarrow \tau_2 \mid \rho.\tau \\
\sigma & ::= & \forall \alpha_1 \cdots \alpha_n.\tau
\end{array}
$$

CONST:
$$\frac{typeof(c) \geq \phi.\tau}{TE \vdash c : \tau, \phi}$$

PRIMAPP:
$$\frac{\begin{array}{c} typeof(PF^n) \geq \phi.(\tau_1 \rightarrow \cdots \tau_n \rightarrow \tau_{n+1}), \phi \\ TE \vdash SE_i : \tau_i, \rho_i \qquad 1 \leq i \leq n \end{array}}{TE \vdash PF^n\ (SE_1, \ldots, SE_n) : \tau_{n+1}, \bigcup_{1 \leq i \leq n} \rho_i}$$

CASE:
$$\frac{\begin{array}{c} TE \vdash SE : (T\ \tau_1 \cdots \tau_n), \rho_0 \\ TE \vdash E_i : \tau, \rho_i \qquad 1 \leq i \leq m \end{array}}{TE \vdash \mathtt{Case}^m\_T\ SE\ (E_i \cdots E_m) : \tau, \bigcup_{0 \leq i \leq m} \rho_i}$$

IDENT:
$$\frac{TE(x) \geq \rho.\tau}{TE \vdash x : \tau, \rho}$$

APP:
$$\frac{TE \vdash SE_1 : (\tau' \rightarrow \rho.\tau), \rho_1 \qquad TE \vdash SE_2 : \tau', \rho_2}{TE \vdash SE_1\ SE_2 : \tau, (\rho \cup \rho_1 \cup \rho_2)}$$

ABS:
$$\frac{\begin{array}{c} TE + \{x_1 \mapsto \tau_1, \ldots, x_n \mapsto \tau_n\} \vdash E : \tau_{n+1}, \rho \\ \text{Let } TM \text{ be the type-map of } \lambda x_1 \cdots x_n.E \\ \begin{array}{rcll} \rho_0 & = & \mathcal{F}(TM) \setminus \mathcal{F}(\tau_1 \rightarrow \cdots \tau_n \rightarrow \tau_{n+1})) & \\ \rho_i & = & \mathcal{F}(\tau_i) \setminus \mathcal{F}(\tau_{i+1} \rightarrow \cdots \tau_n \rightarrow \tau_{n+1}) & 1 \leq i < n \\ \rho' & = & \mathcal{F}(\rho) \setminus (\rho_0 \cup \cdots \cup \rho_{n-1}) & \end{array} \end{array}}{TE \vdash \lambda x_1 \cdots x_n.E : \rho_0.(\tau_1 \rightarrow \rho_1.(\tau_2 \rightarrow \cdots \rho_{n-1}.(\tau_n \rightarrow \rho'.\tau_{n+1}) \cdots)), \phi}$$

BLOCK:
$$\frac{\begin{array}{c} \left.\begin{array}{c} TE + \{x_i \mapsto \tau_i\} \vdash E_i : \tau_i, \rho_i \\ TE_{b_1} = TE + \{x_i \mapsto Gen(TE, \tau_i)\} \end{array}\right\} \quad i \in b_1 \\ \left.\begin{array}{c} TE_{b_1} + \{x_i \mapsto \tau_i\} \vdash E_i : \tau_i, \rho_i \\ TE_{b_2} = TE_{b_1} + \{x_i \mapsto Gen(TE_{b_1}, \tau_i)\} \end{array}\right\} \quad i \in b_2 \\ \cdots \\ TE_{b_k} \vdash SE : \tau_0, \rho_0 \end{array}}{TE \vdash \{\ x_1 = E_1; \cdots; x_n = E_n\ \mathtt{in}\ SE\} : \tau_0, \bigcup_{0 \leq i \leq n} \rho_i}$$

Figure 7.2: Rules for computing Non-Conserved Type Information for Kernel Id Programs.

types as usual. We derive typing judgments of the following form:

$$TE \vdash E : \tau, \rho$$

Here, TE is a type environment mapping identifiers to type schemes, τ is the type assigned to the expression E, and ρ is the set of type-hint instantiations within E that are needed during its type reconstruction. These type-hints are required when a non-type-conserving function is referenced or is applied inside the expression E. All such type instantiations are collected and propagated up to the nearest enclosing function definition where they become part of that function's type-hint requirements.

Looking at Figure 7.2, predefined constants and primitive functions (CONST and PRIMAPP rule) do not give rise to any non-conserved type-variables since they always execute within the current activation frame and never create any partial applications. The CASE rule is also straightforward. It simply collects the type-hint instantiations inductively from its sub-expressions while ensuring that all branches have the same type.

The IDENT rule instantiates the augmented type of a user-defined function, exposing the exact instantiations of its non-conserved type-variables that need to be provided at that point. The augmented type instantiation is immediately split into the actual type τ and the set of type-hint instantiations ρ. Note that the size of the set ρ remains fixed during its type instantiation. In particular, an empty set of type-variables ϕ can never be instantiated to a non-empty set of type instantiations and vice-versa.[2]

New type instantiations may also be introduced by the APP rule, where the augmented type of the result closure exposes the exact instantiations of the non-conserved type-variables at that application site. All such instantiations are collected and propagated to be resolved at the nearest enclosing λ-expression.

The ABS rule computes the set of non-conserved type-variables of a λ-expression and records them within its augmented type so that they may be instantiated later by the IDENT rule or the APP rule. The type-hint sets $\rho_0 \cdots \rho_{n-1}$ are computed for each argument position of the function as given by Definition 6.2. These sets are placed along the type signature of the function just after the argument position where that type information would otherwise be lost. Type-variables that are conserved at the various argument positions are excluded from the corresponding type-hint sets. The final type-hint set ρ' computes the additional type-variables for which type-hints are required by internal sub-expressions of the λ-body.

The BLOCK rule is a generalization of the usual Hindley/Milner LET rule as applied to the more complex syntax of the Kernel Id language. The type generalization operation $Gen(TE, \tau)$ generalizes the augmented type τ (which may contain embedded type-hint sets) into a type scheme $\forall \alpha_1 \cdots \alpha_n . \tau$. We assume that the bindings in a block, numbered $1 \ldots n$, are partitioned into k groups of mutually recursive bindings $b_1 \cdots b_k$ ($b_1 + \cdots + b_k = n$), and these groups are topologically sorted such that definitions occur before their uses. Each group of mutually recursive bindings is type-checked within a type environment that assigns polymorphic type schemes to the identifiers bound in previous groups and monomorphic types to the identifiers bound within the same group. This transformation maximizes Hindley/Milner polymorphism for an unordered sequence of bindings [Gup90, HWe90].

[2] This property ensures that each type-variable instantiation may be treated as an independent parameter to be inserted at that site during translation, although it may introduce some subtle typing discrepancies as discussed in Section 7.2.4.

7.2.2 Type Inference

The type system shown in Figure 7.2 may be directly used as a basis for automatically inferring augmented Hindley/Milner types along the lines of the standard Hindley/Milner type inference algorithm [Mil78]. The type-hint sets are considered to be ordered and of fixed size, and may be treated as part of the type signature of a function. In particular, note that a non-empty set can never be unified with an empty set. Therefore, the usual structural term unification algorithm [Rob65] would suffice for matching types.[3]

The type inference algorithm would be similar to the *infer* algorithm cited in Section 3.4 with minor modifications. We need to do some book-keeping in order to collect and propagate type-hint instantiations from within expressions and process them at the enclosing function definition. The modified type inference algorithm would return the possibly augmented Hindley/Milner type (τ) of an expression along with the set of type-hint instantiations (ρ) gathered from within the expression. Type generalization, instantiation, and substitution would now take place on augmented types. In case of a user-defined function, the algorithm would also compute its *type-map* as given by Definition 6.1 and the type-hint sets $\rho_0 \cdots \rho_{n-1}, \rho'$ as shown in the ABS rule. These sets would then be attached to their appropriate argument positions within the type signature of the function.

7.2.3 Program Translation and Type-Hint Generation

The final step in the compilation process is to add explicit parameters to functions with non-trivial augmented types and to provide appropriate type-hints at their application sites.

Generation of type-hints uses a run-time encoding and decoding scheme as shown in Figure 7.3. The encoding is performed under a Translation Environment Γ that maps free type-variables of a given type τ to value-domain identifiers encoding those type-variables. The encoding scheme **TEnc**$[\![\,]\!]$ produces a *Kernel Id expression* which when executed at run-time generates the type-hint encoding for the given type scheme; it does not generate the encoded type scheme itself. This is so because the encoding scheme is used as part of the source-to-source compilation process that translates a Kernel Id program into another Kernel Id program with explicit type-hint propagation.

For each type constructor T^n, we denote its run-time encoding by a new constant $\overline{T^n}$. A bound type-variable α_i in a type scheme $\forall\alpha_1 \cdots \alpha_n.\tau$ is encoded as a special constant type-constructor $\overline{T^0_{\alpha_i}}$. A family of Kernel Id primitive functions $pack^n$ with arity n are used to pack an encoded type constructor and its arguments into a run-time data-structure.

The decoding scheme **TDec**$[\![\,]\!]$ is used at run-time to convert the encoded type-hints into actual type schemes used during run-time type reconstruction. Although this mechanism is described as the logical inverse of encoding type schemes, the actual decoding format depends on the data format used within the run-time system for type reconstruction.[4]

The program translation and hint generation scheme **TExp**$[\![\,]\!]$ is shown in Figure 7.4. This translation is guided by the typing judgments derived from the typing rules shown in Figure 7.2. The translation rules operate under a Translation Environment Γ that maps free non-conserved type-variables of a function definition to its type-hint parameters.

[3]The careful reader might note that performing structural type matching on the type-hint sets may reject some programs that would be considered to be type-correct in the original Hindley/Milner type system without such sets. We will discuss this issue in Section 7.2.4.

[4]In our current implementation discussed in Chapter 8, the data format used for encoding type-hints is the same as that used within the run-time system for type reconstruction, therefore no decoding is necessary.

$$
\begin{array}{rcl}
\Gamma & \in & \text{Translation Environment} = \text{Type-Variable} \rightarrow \text{Identifier} \\
\overline{\sigma} & \in & \text{Encoded Type Scheme} \\
\mathbf{TEnc}[\![]\!] & \in & \text{Type Scheme} \rightarrow \text{Translation Environment} \rightarrow \textsc{Expression} \\
\mathbf{TDec}[\![]\!] & \in & \text{Encoded Type Scheme} \rightarrow \text{Type Scheme}
\end{array}
$$

Type Scheme Encoding

$$
\mathbf{TEnc}[\![\alpha]\!] \ \ \Gamma \quad = \quad \Gamma(\alpha)
$$

Let $z, z_1, \ldots, z_n$ be new identifiers,

$$
\begin{array}{lcl}
\mathbf{TEnc}[\![(T^n\ \tau_1 \cdots \tau_n)]\!] \ \ \Gamma & = & \{\ \ z_1 = (\mathbf{TEnc}[\![\tau_1]\!]\ \Gamma); \\
& & \quad \cdots \\
& & \quad z_n = (\mathbf{TEnc}[\![\tau_n]\!]\ \Gamma); \\
& & \quad z = \mathit{pack}^{n+1}(\overline{T^n}, z_1, \ldots, z_n); \\
& & \quad \texttt{in}\ z\ \} \\
\mathbf{TEnc}[\![\forall \alpha_1 \cdots \alpha_n.\tau]\!] \ \ \Gamma & = & \mathbf{TEnc}[\![\tau]\!]\ (\Gamma + \{\alpha_i \mapsto \overline{T^0_{\alpha_i}}\}) \qquad 1 \leq i \leq n
\end{array}
$$

Type Scheme Decoding

$$
\begin{array}{lcl}
\mathbf{TDec}[\![\overline{\sigma}]\!] & = & \forall \alpha_1 \cdots \alpha_n.\mathbf{TDec'}[\![\overline{\sigma}]\!] \\
\text{where} \ \ \{\alpha_1, \ldots, \alpha_n\} & = & \mathcal{F}(\mathbf{TDec'}[\![\overline{\sigma}]\!])
\end{array}
$$

$$
\begin{array}{lcl}
\mathbf{TDec'}[\![\overline{T^0_\alpha}]\!] & = & \alpha \\
\mathbf{TDec'}[\![\overline{(\overline{T^n}, \overline{\tau_1}, \ldots, \overline{\tau_n})}]\!] & = & (T^n\ \mathbf{TDec'}[\![\overline{\tau_1}]\!]\ \cdots\ \mathbf{TDec'}[\![\overline{\tau_n}]\!])
\end{array}
$$

Figure 7.3: Encoding and Decoding of Type Schemes.

Most of the translation rules are straightforward. Constants do not require any translation. The rules for primitive application, `Case`-expression, and block recursively translate their subexpressions.

The translation of a function identifier converts the exact instantiations of its non-conserved type-variables into explicit type-hint arguments using the encoding shown in Figure 7.3. Similarly, the translation of a function application inserts appropriate type-hints at that application site as directed by the function signature.

The translation of a λ-expression adds explicit hint parameters $y_1 \cdots y_m$ at the appropriate position corresponding to each non-conserved type-variable obtained from its typing judgment. We also record this mapping as the *hint-map* of the λ-expression and use it to extend the translation environment for the body of the given λ-expression.

We assume that the type-map (Definition 6.1) of a λ-expression is updated to reflect the new type-hint parameters that are added to its type signature and the new local bindings that are created within its body during the translation. This change does not affect the set of free type-variables of the type-map because encoded type-hints have a fixed, pre-defined non-polymorphic type, and the types for all other additional identifier bindings are already present within the type-map.

After this program transformation, all the type information needed to fully instantiate the type-map of a function is available at run-time within its function activation frame, either directly as run-time type-hints or indirectly *via* instantiations of conserved type-variables in its type-map. In the next section, we will show a type reconstruction algorithm that uses this information at run-time to reconstruct the complete dynamic state of the machine.

$$\begin{array}{rcl}\Gamma & \in & \text{Translation Environment} = \text{Type-Variable} \rightarrow \text{Identifier}\\ \mathbf{TExp}[\![\,]\!] & \in & \textsc{Expression} \rightarrow \text{Translation Environment} \rightarrow \textsc{Expression}\end{array}$$

CONST:

$$\mathbf{TExp}[\![c]\!]\ \ \Gamma \quad = \quad c$$

PRIMAPP: Let $z, z_1, \ldots, z_n$ be new identifiers,

$$\begin{array}{lcl}\mathbf{TExp}[\![PF^n\ SE_1 \cdots SE_n]\!]\ \ \Gamma & = & \{\ z_1 = (\mathbf{TExp}[\![SE_1]\!]\ \Gamma);\ \cdots\\ & & \quad z_n = (\mathbf{TExp}[\![SE_n]\!]\ \Gamma);\\ & & \quad z = PF^n\ z_1\ \cdots\ z_n;\\ & & \quad \texttt{in}\ z\ \}\end{array}$$

CASE: Let z, z_0 be new identifiers,

$$\begin{array}{lcl}\mathbf{TExp}[\![\texttt{Case}^m\_T\ SE\ (E_1 \cdots E_m)]\!]\ \ \Gamma & = & \{\ z_0 = (\mathbf{TExp}[\![SE]\!]\ \Gamma);\\ & & \quad z = \texttt{Case}^m\_T\ z_0\ (\ (\mathbf{TExp}[\![SE_1]\!]\ \Gamma)\ \cdots\\ & & \qquad\qquad (\mathbf{TExp}[\![SE_n]\!]\ \Gamma)\ );\\ & & \quad \texttt{in}\ z\ \}\end{array}$$

IDENT: Given typing judgment $TE \vdash x : \tau, \rho$ where $\rho = \{\tau_1 \cdots \tau_n\}$,
Let $z, z_1, \ldots, z_n$ be new identifiers,

$$\begin{array}{lcl}\mathbf{TExp}[\![x]\!]\ \ \Gamma & = & \{\ z_1 = (\mathbf{TEnc}[\![\tau_1]\!]\ \Gamma);\ \cdots\\ & & \quad z_n = (\mathbf{TEnc}[\![\tau_n]\!]\ \Gamma);\\ & & \quad z = x\ z_1\ \cdots\ z_n;\\ & & \quad \texttt{in}\ z\ \}\end{array}$$

APP: Given typing judgment $TE \vdash SE_1\ SE_2 : \tau, (\rho \cup \rho_1 \cup \rho_2)$ where $\rho = \{\tau_1 \cdots \tau_n\}$,
Let $z, z', z'', z_1, \ldots, z_n$ be new identifiers,

$$\begin{array}{lcl}\mathbf{TExp}[\![SE_1\ SE_2]\!]\ \ \Gamma & = & \{\ z' = (\mathbf{TExp}[\![SE_1]\!]\ \Gamma);\\ & & \quad z'' = (\mathbf{TExp}[\![SE_2]\!]\ \Gamma);\\ & & \quad z_1 = (\mathbf{TEnc}[\![\tau_1]\!]\ \Gamma);\ \cdots\\ & & \quad z_n = (\mathbf{TEnc}[\![\tau_n]\!]\ \Gamma);\\ & & \quad z = z'\ z''\ \ z_1\ \cdots\ z_n;\\ & & \quad \texttt{in}\ z\ \}\end{array}$$

ABS: Given typing judgment $TE \vdash \lambda x_1 \cdots x_n.E : \rho_0.(\tau_1 \rightarrow \cdots \rho_{n-1}.(\tau_n \rightarrow \rho'.\tau_{n+1}) \cdots), \phi$
where $(\rho_0 \cup \cdots \cup \rho_{n-1} \cup \rho') = \{\alpha_1 \cdots \alpha_m\}$,
Let $y_1, \ldots, y_m$ be new parameters with hint-map $HM = \{\alpha_1 \mapsto y_1, \ldots, \alpha_m \mapsto y_m\}$,

$$\mathbf{TExp}[\![\lambda x_1 \cdots x_n.E]\!]\ \ \Gamma \quad = \quad \lambda y_{\rho_0} x_1 y_{\rho_1} \cdots y_{\rho_{n-1}} x_n y_{\rho'}.\ (\mathbf{TExp}[\![E]\!]\ (\Gamma + HM))$$

BLOCK: Let z be a new identifier,

$$\begin{array}{lcl}\mathbf{TExp}[\![\{x_1 = E_1; \cdots; x_n = E_n\ \texttt{in}\ SE\}]\!]\ \ \Gamma & = & \{\ x_1 = (\mathbf{TExp}[\![E_1]\!]\ \Gamma);\ \cdots\\ & & \quad x_n = (\mathbf{TExp}[\![E_n]\!]\ \Gamma);\\ & & \quad z = (\mathbf{TExp}[\![SE]\!]\ \Gamma);\\ & & \quad \texttt{in}\ z\ \}\end{array}$$

Figure 7.4: Program Translation and Hint Generation Rules.

7.2.4 Discussion

Type Mismatch in Curried Functions

The augmented type system presented in Section 7.2.1 above is a straightforward modification of the standard Hindley/Milner type system, but it has one drawback that it restricts the set of type-correct programs to those that are also type-reconstructible. In particular, this system may reject some programs that would have been considered type-correct in the usual Hindley/Milner type system without any type-hint sets. The following example illustrates this point:

Example 7.1:

```
def f1 x y = y;                  % f1 :: ∀t0t1.φ.(t0 → {t0}.(t1 → φ.t1))

def f2 x =                       % f2 :: ∀t0t1.φ.(t0 → φ.(t1 → φ.t1))
  { def h2 y = y;
    in h2 };

g1 = (if ... then f1 else f2);              % Static Type Error!
g2 = if ... then f1 1 ⟨int⟩ else f2 1;      % No Static Type Error.
```

The functions `f1` and `f2` have the same type signature in the usual Hindley/Milner type system but they have different type signatures in the current system because `f1` requires a type-hint for its first argument, while `f2` does not. This is because the type of the first argument of the function `f1` is not conserved according to Definition 6.2, while both `f2` and the internal function `h2` are considered to be type-conserving. This type mismatch shows up in the binding for `g1` which is flagged as a type-error in our augmented type system. However, the binding for `g2` may be typed without any problem because the type-hint required by the function `f1` has been already inserted.

The above example shows that our type system makes a subtle distinction between implicitly curried multi-arity functions such as `f1` and their explicitly curried counterparts such as `f2`. To be precise, this difference shows up only in non-type-conserving functions as shown in the above example; type-conserving functions would always have empty type-hint sets. This difference exposes an important run-time characteristic of such functions: the number of applications after which a function closure expands into an activation frame, which is controlled by their syntactic arity and not by their semantic typing.

In a way, this difference is to be expected because `f1` and `f2` carry different objects within the closures resulting from their first application and hence require different amount of type reconstruction information. Since `f1` is implicitly curried, it simply records its first argument within its closure. This forces the type conservation mechanism (Definition 6.2) to insert additional type information in order to ensure subsequent type reconstruction of this argument even if it was never used within the function's body. On the other hand, `f2` produces an entirely new closure `h2` on its first application that is completely independent of its first argument. So, there is no need to preserve its type within the returned closure.

It should be noted that the type mismatch between `f1` and `f2` is generated not merely because we have chosen to represent the type-hint information explicitly within the type signatures of these functions. This type mismatch is actually a consequence of the underlying compilation mechanism that treats additional type-hints just like any other function parameters. In particular, it would not be possible to correctly compile the binding for `g1` even if the type-hint analysis was done after the type-checking phase. This is because under our current compilation strategy, only `f1` requires a type-hint which is determined only after it is applied to a particular argument.

Multiple Type Signatures

Another interesting difference between this type system and the usual Hindley/Milner type system shows up with higher-order functions that take other functions as arguments. Consider the `map` function shown earlier in Example 6.1 which is reproduced below:

Example 7.2:

```
def map f nil = nil        % map :: ∀t0t1.φ.(t0 → φ.t1) → φ.((list t0) → φ.(list t1))
 |  map f (y:ys) =         % map :: ∀t0t1.φ.(t0 → {t0}.t1) → φ.((list t0) → {t0}.(list t1))
      (f y):(map f ys);

def enlist x = x:nil;
g1 = map enlist (1:nil);          % No type-hint needed.

def ignore x y = y;
g2 = map ignore (1:nil) ⟨int⟩;    % Type-hint needed internally by ignore.
```

Two possible types for the `map` function are shown. The first type assumes that the input function `f` is type-conserving and therefore would not need any type-hints when it is applied within the body of `map`. This permits type-conserving functions such as `enlist` to be passes to `map` as usual. The second type signature assumes that the incoming function would not be type-conserving and would need a type-hint at its application within the body of `map`. This type-hint propagates up to the definition of the `map` function and shows up in its type signature after the second argument. This allows non-type-conserving functions such as `ignore` to be passed as arguments to `map`.

It may be a little disconcerting to note that the `map` function no longer has a single type. On the other hand, the two versions of `map` are truly different functions and must be compiled as such—one that propagates type-hints and the other that does not. One can think of the original Hindley/Milner type signature of the `map` function as being *overloaded* with the various intended versions. The compiler may selectively produce these specialized versions according to the type of the arguments supplied to `map`.

Alternate Compilation Scheme

Both the problems presented above may be fixed by making the type-hint compilation more uniform and transparent to the standard parameter passing mechanism. In this section, we briefly examine one such compilation scheme.

Instead of inserting type-hints required at a given argument position as additional parameters, we may put them in a separate record and pass a single pointer to that record to a fixed entry point identified by that argument position. Effectively, this adds one additional parameter for every argument position whether or not any type-hints are needed at that position. The advantage of this compilation scheme is that it completely dissociates propagation of type-hints from regular parameter passing, although it takes additional frame space and time overhead in allocating type-hint records. In this scheme, Empty type-hint records need not be propagated at all, while non-empty type-hint records may be passed to even type-conserving functions that do not require this information. The latter property fixes the problem of compiling `g1` shown in Example 7.1. Now, a type-hint record would be created for each application site of `g1` which would be used by `f1` during type reconstruction but would be simply ignored by `f2`.

This scheme also makes the compilation of higher-order functions such as the `map` function of Example 7.2 more uniform. Now the `map` function may be compiled to always propagate the type-hint record it receives from its first argument position to its internal application site. If no actual type-hint record is supplied from outside then this mechanism essentially propagates an empty type-hint record to the internal application site. However, specialized version of `map` that do not pay this overhead may still be compiled as an optimization.

7.3 Run-time Type Reconstruction

7.3.1 Type Reconstruction Requirements

Before we describe our type reconstruction algorithm, we summarize the requirements for full type reconstruction as discussed in previous sections. We use both compile-time and run-time information.

1. The compile-time information consists of the type-map (Definition 6.1), the hint-map (Definition 6.3) and the arity of each function that is stored in the symbol table entry for that function.

2. Furthermore, every function in the program must be transformed as shown in Section 7.2 to propagate explicit type-hints for its non-conserved type-variables.

3. The run-time information consists of the global dynamic environment and the root frame of the activation tree that remain live and are assumed to be accessible throughout the computation (Section 6.2.1). The activation tree hangs from the root activation frame and is modified dynamically, as the program executes, by the procedure linkage code.

4. At program invocation time, complete type information is available for the user query expression and the root activation frame (Section 6.2.1). Therefore, the root frame should already be marked as reconstructed.

5. Given any activation frame, we should be able to identify the function associated with it, its parent activation frame, and the application site in the parent frame that created this activation frame.[5] Typically, the conventional return address information within the callee is sufficient for this purpose.

6. Proper decoding mechanism should exist for types and type schemes encoded as type-hints (Figure 7.3). A run-time mechanism for type unification is also required, although it can be simplified considerably since static type-checking guarantees that unifications performed within the reconstruction algorithm cannot fail.

7.3.2 The Reconstruction Algorithm

Figure 7.5 shows the pseudo-code for the reconstruction algorithm RECONSTRUCT-FRAME which is invoked at run-time to reconstruct the types of all variables in a given activation frame. RECONSTRUCT-FRAME takes an activation frame as a parameter and returns a fully instantiated type-map for that frame. For ease of presentation, the algorithm makes use of several auxiliary functions which we will explain as we go along.

[5]We ignore the issue of "tail calls" whose compilation was discussed in Section 6.5.

RECONSTRUCT-FRAME(activation-frame)

$\triangleright$ *Return if already reconstructed.*

if FRAME-RECONSTRUCTED(activation-frame)

then return FRAME-TYPE-MAP(activation-frame)

$\triangleright$ *Otherwise, start reconstruction.*

activation-fn $\leftarrow$ ACTIVATION-FN(activation-frame)

$\triangleright$ *Copy the function's type-map.*

type-map $\leftarrow$ TYPE-MAP(activation-fn)

$\{\alpha_1, \ldots, \alpha_n\} \leftarrow \mathcal{F}$(type-map)

$S_{copy} \leftarrow \{\alpha_i \mapsto \beta_i\}$ *where* $\beta_1, \ldots, \beta_n$ *are new.*

$\triangleright$ *Process the type-hints.*

hint-map $\leftarrow$ HINT-MAP(activation-fn)

$S_{hint} \leftarrow$ { **forall** $(\alpha \mapsto x)$ **in** hint-map

  $\overline{\sigma} \leftarrow$ FETCH-ARGUMENT(x, activation-frame)

  $\sigma \leftarrow$ **TDec**$[\![\overline{\sigma}]\!]$

  collect $(S_{copy}\alpha \mapsto \sigma)$ }

$\triangleright$ *We are done if the type-map is fully instantiated.*

if $\mathcal{F}(S_{hint}S_{copy}$(type-map)$) = \phi$

then FRAME-TYPE-MAP(activation-frame) $\leftarrow S_{hint}S_{copy}$(type-map)

$\triangleright$ *Otherwise, obtain call site information from the parent.*

else { parent-activation-frame $\leftarrow$ PARENT-ACTIVATION-FRAME(activation-frame)

  parent-type-map $\leftarrow$ RECONSTRUCT-FRAME(parent-activation-frame)

  $\tau_{use} \leftarrow$ USE-TYPE(activation-frame, parent-type-map)

  $\tau_{def} \leftarrow$ DEF-TYPE(activation-fn, S_{copy}(type-map))

  if FULL-APP(activation-frame, parent-type-map)

  then $S_{def\text{-}use} \leftarrow$ UNIFY(τ_{def}, τ_{use})

  else { $k \leftarrow$ ARITY(activation-fn)

    $\tau_1 \rightarrow \cdots \tau_k \rightarrow \tau_{k+1} \leftarrow \tau_{def}$

    $S_{def\text{-}use} \leftarrow$ UNIFY$(\tau_k \rightarrow \tau_{k+1}, \tau_{use})$ }

  FRAME-TYPE-MAP(activation-frame) $\leftarrow S_{def\text{-}use}S_{hint}S_{copy}$(type-map) }

FRAME-RECONSTRUCTED(activation-frame) $\leftarrow$ **true**

return FRAME-TYPE-MAP(activation-frame)

Figure 7.5: The Type Reconstruction Algorithm.

RECONSTRUCT-FRAME is divided into several sections. We begin at Line 1 by checking if the given activation frame has already been reconstructed. If so, the previously recorded frame type-map is returned immediately. Otherwise, we initiate the reconstruction process.

The first section, Lines 3–6, initializes the data-structures used in the reconstruction. We extract the name of the current activation function from the given frame using the selector function ACTIVATION-FN and instantiate its type-map with fresh type-variables by building a type substitution S_{copy} for its free type-variables. This is necessary so that types from multiple activations of the same polymorphic function do not inadvertently interfere with each other.

The next section, Lines 7–11, builds a type substitution S_{hint} for all the non-conserved type-variables of the function as prescribed by its hint-map. The type-hint corresponding to each hint parameter present in the hint-map is fetched from the activation frame and then decoded according to Figure 7.3. The resulting type schemes are the run-time instantiations of the non-conserved type-variables present in the hint-map.

Following this, Line 12 checks to see if all free type-variables of the type-map have been instantiated to either ground or polymorphic types. If so, the reconstruction is complete and the fully instantiated type-map is recorded at Line 13. The FRAME-RECONSTRUCTED flag is set at Line 24 and the reconstructed type-map is returned at Line 25.

If the test fails at Line 12, Lines 14–22 obtain the remaining information from the activation tree as follows. First, the type-map of the parent of the current activation is reconstructed by calling RECONSTRUCT-FRAME recursively with the parent's activation frame. Using this type-map and the current activation, the auxiliary function USE-TYPE obtains the reconstructed type-instance of the call site responsible for invoking the current function (see item 5 of Definition 6.1). This type-instance, τ_{use}, is then unified with the defined type of the current function, τ_{def} that is available within the current type-map. This unification fully instantiates all the remaining type-variables in the current type-map which is recorded at Line 23 and is returned at Line 25 as before.

The matching of τ_{def} to τ_{use} is slightly complicated by the fact that the current activation could either be a full application of a k-arity function to all its arguments or it could simply be the final (k-th) application of a curried function closure that has already accumulated $k-1$ arguments in previous partial applications. The recorded application site type instance τ_{use} would be different in these two cases and therefore it must be properly aligned before matching with the function's full type signature τ_{def}.[6] This application information is also recorded within the parent's type-map and is obtained at Line 18 using the auxiliary function FULL-APP. In case of a full application, τ_{use} is directly unified with τ_{def} recorded in the current function's type-map at Line 19. In case of a curried application, τ_{use} must be unified with just the final application type $\tau_k \rightarrow \tau_{k+1}$ of the defined type τ_{def} as shown at Line 22.

7.3.3 Reconstruction Complexity

A few observations about the reconstruction algorithm are worth pointing out. First, the entire activation frame of a function is reconstructed at once. This is possible because the types of all the objects present in an activation frame share the same set of free type-variables which are precisely captured and instantiated using its type-map. This obviates the need to traverse the activation tree multiple times in order to reconstruct the types of various identifiers belonging to the same frame.

Second, we cache the reconstructed type-maps of all activation frames for future references by their child frames. Therefore, no activation frame may need to be reconstructed more than

[6]In our earlier paper [AC93], this operation was abstracted into the auxiliary function UNIFY-ALIGNED.

once. Furthermore, since the root frame is already marked as reconstructed at the start of the program, the algorithm is guaranteed to terminate properly as it recursively climbs the activation tree at Line 15.

Finally, the algorithm climbs the activation tree from the current activation frame only as far up as necessary. The climbing process terminates at the first ancestor frame that has already been reconstructed, or earlier if sufficient information is available *via* the type-hints. This avoids traversing the activation tree from the root activation frame to all its leaves as suggested in [Gol91] which would involve reconstructing all the activation frames within the dynamic activation tree. Our algorithm pays only incremental cost for each request for reconstruction, which is a very useful feature for interactive applications such as a source debugger.

The cost of the algorithm RECONSTRUCT-FRAME shown in Figure 7.5 depends on the following factors:

1. The number of ancestor frames reconstructed due to recursive calls to the algorithm RECONSTRUCT-FRAME at Line 15.

2. The cost of decoding the type-hints at Lines 7–11.

3. The cost of unification at Line 19 or Line 22.

The maximum number of ancestor frames reconstructed in a given call to the algorithm RECONSTRUCT-FRAME is bounded by the number of frames occurring between the current activation frame and the root frame. In a sequential system, this is all the frames sitting on the stack. In a parallel system, this is the number of frames on any path from a leaf to the root in the dynamic activation tree which is only the *depth* of the dynamic activation tree and not its overall *size*. Of course, since all reconstructed type-maps are cached, the overall cost of reconstructing every frame within the dynamic activation tree is still linear in the total number of activation frames, assuming a unit cost for type unification and type-hint decoding.

The cost of decoding the type-hints depends on the number of non-conserved type-variables in the type-map and the size of their run-time type instantiations. Similarly, the cost of unification is proportional to the size of the function's instantiated type. Although it is possible to write functions whose Hindley/Milner types are exponentially large compared to the size of the function itself [Mai90], such cases are rare. Typically, functions possess small type signatures that can be efficiently manipulated using graphical representations. Non-conserving functions are rare as well and run-time type instantiations of non-conserving type-variables are also small.

The interesting observation here is that the cost of reconstructing a type-map for a given activation frame does not depend on the number of slots in the activation frame or the total size of the type-map itself, but only on the size of the type signature of the corresponding activation function. This is because we never need to examine or copy the type of every identifier recorded in the type-map during its reconstruction. We only instantiate its free type-variables.[7]

7.4 Correctness of the Type Reconstruction Algorithm

In this section, we will show that the type reconstruction algorithm given in Figure 7.5 is correct, *i.e.*, it infers the exact type for every object at any time during the execution of a program. We will define the notion of the *exact* type for an object shortly, but for the time being it may be viewed as the type that would have been attached to the object had we computed and

[7] An independent program such as a debugger or a garbage collector may ultimately need to examine the reconstructed types of every element in the activation frame. That cost is not included in reconstruction.

propagated source type information all through the execution of the program. In dynamically-typed languages such as Lisp, this is exactly how dynamic type-checking is performed. Every object is *tagged* with its type and that information is carried through each computation step. The type of every new object (including scalars such as integers and floats) is computed along with its value and is attached to the value as its tag. Of course, computing types is a substantial overhead during program execution which is why we have chosen to perform dynamic type reconstruction instead of dynamic type maintenance.

The Kernel Id language (Figure 7.1), its run-time execution model (Figure 6.2), and the type reconstruction algorithm (Figure 7.5) are all quite complex. In order to be able to argue about the correctness of the algorithm, we make several theoretical simplifications. These simplifications allow us to model these concepts cleanly and distill the basic characteristics of the reconstruction algorithm.

7.4.1 Simple Expression Language and its Semantic Model

As the first step, we restrict ourselves to the simple expression language described in Chapter 3. This is because we have already made a considerable effort to rigorously define the static and dynamic semantics for this language. We already have an operational semantic model for this language (Definition 3.12) and we have shown the consistency between the static and the dynamic semantics (Theorem 3.16). This consistency is the main tool using which we will show the correctness of our type reconstruction algorithm. It may be noted that the problem of complete type reconstruction is independent of the issue of parallel or sequential execution. Therefore, restricting ourselves to a strict, sequential language instead of dealing with the fully parallel execution model of Id does not affect the reconstruction algorithm or the issue of its correctness.

It is easy to see the correspondence between the Kernel Id language shown in Figure 7.1 and the simple expression language shown in Section 3.1.1. Most Kernel Id expressions have direct analogue in the simple expression language. The important simplifications are that mutually recursive functions must be combined into a single self-recursive function, `Case`-expressions must be broken up into a series of conditional expressions and blocks must be converted into nested `let`-bindings of mutually recursive definitions.

7.4.2 Partial Execution and the Dynamic Activation Tree

The second step is to model the state of the machine at the moment when type reconstruction is requested. In the relational formulation of the dynamic semantics shown in Section 3.1.2, an evaluation of a top-level query expression may be described by a logical derivation tree of evaluation judgments of the following form:[8]

$$e \vdash a/s \Rightarrow v/s'$$

The evaluation derivation tree for the top-level query provides a logical proof of how evaluations of sub-expressions contribute towards the final result of the entire program according to the dynamic inference rules shown in Figure 3.1. We will now treat this derivation tree of evaluation judgment relations as representing the computation itself. The complete derivation tree for the top-level query corresponds to the entire program computation. Each judgment

[8]We assume that the result of the evaluation is not `err`. This is because we assume that the entire program including the top-level query expression is type-correct. Therefore, by the Soundness Theorem 3.16, the evaluation can never run into a run-time type-error.

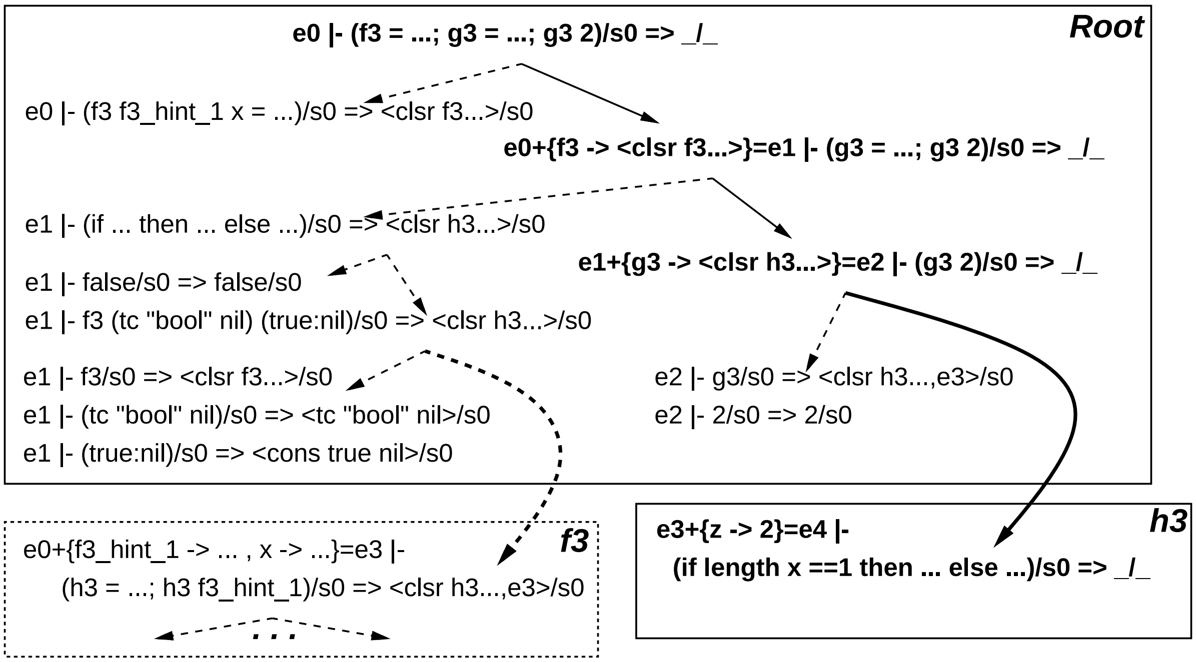

Figure 7.6: The Evaluation Derivation Tree for Example 7.3.

in this tree may be considered as a providing a *place-holder* for the initial store and the final result (value and a new store) computed within that judgment. The store is sequentialized through the entire tree in a predictable depth-first fashion, while the values propagate from the leaves of the tree towards the root—the value of the top-level query being the value of the whole computation. Values may also be passed from one branch of the tree to the other *via* the environment.

The overall process of evaluation may be viewed as a step-wise unfolding of the evaluation derivation tree. We start with the top-level query evaluation judgment using the initial dynamic environment, the initial store, and an empty place-holder for the result. In order to compute the overall result, the top-level evaluation judgment unfolds into a set of antecedent judgments needed by the dynamic inference rule that is selected according to the immediate structure of the query expression. Each such unfolding creates empty place-holders for the results of intermediate evaluation judgments. On reaching the leaves, values are created spontaneously using CONST, IDENT, or ABS rules, and are used to fill the place-holders for the leaf judgments. On each successive computation step, these values fill the place-holders of their parent judgments, until they reach an inference rule with multiple antecedents such as APP, TUPLE, or LET rules, in which case a new sub-tree of evaluation judgments is spawned.

As an example of this process, consider the computation shown in Example 6.8 which is reproduced below:

Example 7.3:

```
def f3 f3_hint_1 x =
  { def h3 h3_hint_1 z = if length x == 1
                          then z:nil
```

```
                              else z:z:nil;
      in h3 f3_hint_1 };
   g3 = if ...
        then f3 (tc "int" nil) (1:nil)
        else f3 (tc "bool" nil) (true:nil);
   g3 2;
```

This program has been translated according to the scheme presented in Section 7.2 with the appropriate type-hints added. The evaluation derivation tree for this computation is depicted in Figure 7.6. Each incomplete evaluation judgment in the derivation tree is expanded downwards into its antecedent judgments according to the dynamic inference rules of Figure 3.1. Not all branches of the derivation tree have been expanded yet. An empty place-holder (_) is used to represent an unknown value or a store within incomplete or unexplored judgments. In addition, we also collapse the sub-trees that have been fully evaluated (shown in light typeface) up to the highest completed evaluation judgment.

Such a partially expanded evaluation derivation tree may be used to model the exact state of a computation at any given point in time:

Definition 7.1 (Partial Execution Tree) *A partial execution tree is a structural tree-prefix*[9] *of the complete evaluation derivation tree for the top-level query expression with the following characteristics:*

1. *Each node consists of a possibly incomplete evaluation judgment of the form* $e \vdash a/s \Rightarrow v/s'$.
2. *Sub-trees consisting entirely of complete evaluation judgments are collapsed into a leaf judgment* $e \vdash a/s \Rightarrow v/s'$ *corresponding to the highest evaluation judgment that has received its value. These nodes represent terminated computation.*
3. *Internal evaluation judgments* $e \vdash a/s \Rightarrow \_/\_$ *that have been expanded but not yet fully evaluated contain empty place-holders (_) to receive their values. These nodes represent the active machine state.*
4. *Unexpanded judgments* $e \vdash a/\_ \Rightarrow \_/\_$ *are also represented by a leaf with empty place-holders. These nodes represent the computation to be spawned in the future.*

Note that if we model the store independently as an external data-structure rather than threading it sequentially through the judgments, we can model parallel computation within this framework by spawning several branches of the partial execution tree in parallel. The only modification needed in the current dynamic semantics to model this situation would be to use a least-upper-bound ($\sqcup$) operation on stores that would combine stores from various branches of the execution tree into a single store.

It is useful to draw a correspondence between the actual dynamic activation tree at any given time during the execution of a program and its partial execution tree as described above. This may be seen by comparing Figure 6.6 that shows the dynamic activation tree for Example 7.3 with Figure 7.6 that shows its partial execution tree. The following correspondences emerge:

[9] A sub-tree starting at the root of the original tree with some of its branches clipped is called a *tree-prefix* of the tree.

1. The root frame of the dynamic activation tree corresponds to the root judgment of the partial execution tree which is evaluating the top-level query expression provided by the user.

2. The type of the query expression is completely known at the beginning of the computation which corresponds to the fact that the root frame in the dynamic activation tree is always marked as reconstructed.

3. Each activation frame present in the dynamic activation tree corresponds to a subset of evaluation judgments within the partial execution tree that belong to the body of the applied function and hang from its application evaluation judgment. In other words, each application evaluation judgment within the partial execution tree may be viewed as initiating a new activation frame for the applied function.

4. Collapsing evaluation sub-trees for completed evaluation judgments corresponds to the fact that the activation frames within that branch of the computation have been deallocated and just the final value is available within the current frame.

With the above correspondence in mind, the partial execution tree serves as an accurate logical model of the actual dynamic activation tree.

7.4.3 Type Reconstruction

Given the definition of the dynamic state of the machine as a partial execution tree, type reconstruction may be viewed as the process of computing the exact type of each value present in the partial execution tree at any given time. Using the formal machinery at hand, this corresponds to generating a type derivation tree using the static semantics inference rules shown in Figure 3.2, that parallels the structure of the given partial execution tree. This is captured in the following definition:

Definition 7.2 (Type Reconstruction) *Type reconstruction of a given partial execution tree is defined to be a type derivation tree with the same structure as the partial execution tree with the following characteristics:*

1. *For each evaluation judgment in the partial execution tree of the form $e \vdash a/s \Rightarrow v/s'$, where s, v, and s' may be empty place-holders, the type derivation tree has a corresponding valid elaboration judgment of the form $E \vdash a : \tau$. Furthermore, the type τ is the most general type satisfying this elaboration.*

2. *For each completed evaluation judgment of the form $e \vdash a/s \Rightarrow v/s'$ and the corresponding typing judgment $E \vdash a : \tau$, there exists a store typing S such that $S \models e : E$ and $\models s : S$.*

Using the Soundness Theorem 3.16, the second condition in the above definition immediately allows us to conclude that the computed value v is consistent with the type τ under a suitably constructed new store typing. In addition, the first condition ensures that this is the most general type of the value v. Therefore, this type τ is taken to be the *exact* reconstructed type of the value v.

7.4.4 The Type Reconstruction Algorithm

The reconstruction algorithm shown in Figure 7.5 reconstructs one activation frame at a time, although it may be applied to each frame within the current dynamic activation tree to reconstruct the whole state of the machine. The actual order in which frames are reconstructed is not important, nor is the fact that we cache the reconstructed type-maps. Therefore, we will assume that all frames in the dynamic activation tree are reconstructed in one sweep that starts at the root frame and works its way downwards towards the leaf frames. This does not change the correctness problem because we are interested in showing the correctness of *what* the algorithm computes, not necessarily *how* it computes it.

As shown earlier, we have modeled the dynamic activation tree as a partial execution tree (Definition 7.1), and the process of type reconstruction as constructing a type derivation tree for it (Definition 7.2). Therefore, all we need to do now is to show that our type reconstruction algorithm given in Figure 7.5 indeed constructs a valid, most general type derivation tree according to Definition 7.2. To accomplish this, we need to abstract the reconstruction algorithm in terms of traversing the partial execution tree and constructing the corresponding type derivation tree.

The first observation to be made about the reconstruction algorithm shown in Figure 7.5 is that it reconstructs an entire frame at a time by instantiating the static type-map of the function corresponding to that frame. The static type-map of a function corresponds to the most general, static type derivation tree of its body. This is because the static type-map records the compile-time type of every sub-expression and free identifiers occurring within the body of the function (Definition 6.1). Furthermore, these types are computed using the type inference algorithm *Infer* mentioned in Section 3.4. The soundness of this algorithm (Proposition 3.19) ensures that we can construct a valid type derivation tree for the entire body of the function, while its completeness (Proposition 3.20) ensures that we obtain the most general type for each sub-expression.

Thus, instantiating the static type-map of a function with a substitution can be viewed as instantiating the entire static type derivation tree of the function body with that substitution. The validity of the derivation tree after substitution is ensured by the stability of typing judgments under substitution (Proposition 3.9). Note that the structure of the instantiated type derivation tree matches the portion of the partial execution tree that corresponds to the activation frame being reconstructed. Sub-trees that are completely evaluated and hence have been collapsed to a leaf in the partial execution tree may also be collapsed in the typing derivation tree.

The second observation about the reconstruction algorithm is regarding the construction of the instantiating substitution $S_{def\text{-}use}S_{hint}S_{copy}$ for the callee's type-map. The purpose of this substitution is to fully instantiate the static type-map of the callee according to the types of its actual arguments and the result, so that the corresponding type derivation tree for the callee's body matches the application site in the caller's derivation tree. The two independent components[10] $S_{def\text{-}use}$ and S_{hint} are responsible for two different sets of arguments supplied to the callee. The substitution $S_{def\text{-}use}$ conveys the type instantiation information due to the arguments and the result present at the final application site, while the substitution S_{hint} provides the instantiation information due to the arguments supplied at previous partial application sites.

The compiler support for type-hint generation and propagation (Section 7.2) provides the mechanism by which we make the relevant type information available at the final application

[10]The third component S_{copy} simply serves to make a copy of the type-map and therefore does not provide any additional information.

site. The most important property of this mechanism is type conservation (Definition 6.2) which ensures that the exact type instantiation for every type-variable within a function's type-map is preserved at each of its application sites. For non-conserved type-variables, the type-hint generation and propagation phase described in Section 7.2.3 encodes their dynamic type instantiations at each partial application site and stores them within the returned closure. This ensures that these type instantiations would remain accessible in encoded form even when the computation that produced the closure has terminated. The substitution S_{hint} during type reconstruction represents these type-variable instantiations. For conserved type-variables, the type of the arguments present at the final application site within the type derivation tree of the caller provides their exact instantiation. The substitution component $S_{def\text{-}use}$ captures these instantiations. Type conservation (Definition 6.2) guarantees that together these two substitutions fully instantiate all the type-variables present within the callee's type-map in accordance with the types of the actual arguments and the result of the function application.

As discussed above, the reconstruction algorithm ensures that the instantiated type derivation tree computed for the callee's body matches its application site within the type derivation tree of the caller's body. This process effectively "glues" the instantiated type derivation tree of the callee's body at the APP rule within the caller's type derivation tree, producing a single typing derivation tree that structurally corresponds to the partial execution tree across this application. Below, we formalize the construction of the type derivation tree in the above manner and show its consistency with respect to the current partial execution tree.

7.4.5 Correctness of the Algorithm

We model the entire computation including the initial program loading/linking phase using a partial execution tree given by Definition 7.1. The program loading and linking phase construct the static and the dynamic environment within which the top-level query expression is evaluated. This is not part of the real reconstruction process because it is performed before initiating the execution of the top-level query expression. But, in our theoretical formulation it is simpler to start with empty static and dynamic environments, and an empty store that are consistent with each other by definition.

Each loading/linking step adds a new `let`-binding to the partial execution tree and the type derivation tree, adding its type and value to the static and the dynamic environments respectively. Since each binding is well typed, it follows from the Soundness Theorem 3.16 that we end up in a store typing S_0 such that each top-level binding value is consistent with its corresponding type, and that the static environment E_0, the dynamic environment e_0, and the store s_0 obtained after loading/linking are also consistent:

$$S_0 \models e_0 : E_0 \qquad \text{and} \qquad \models s_0 : S_0 \tag{7.1}$$

Now we are ready to show that the reconstruction algorithm of Figure 7.5 is correct, *i.e.*, given a logical partial execution tree as given by Definition 7.1, it computes the corresponding logical type derivation tree as given by Definition 7.2.

Theorem 7.3 (Correctness of Type Reconstruction) *The reconstruction algorithm shown in Figure 7.5, when applied to the complete dynamic activation tree at any time during program execution, produces the exact types for every value computed until that time.*

Proof: by induction on the size of the partial execution tree (Definition 7.1). Since the top-level query is guaranteed to be well-typed, we start with its type derivation tree. Looking at the static inference rules shown in Figure 3.2 and the dynamic inference rules shown in Figure 3.1, it is clear that the structure of the type derivation tree must correspond to the partial execution tree except possibly at the APP or the ABS rules where the number of judgment antecedents differ between the static and the dynamic rules. We recurse down the partial execution tree and the current type derivation tree simultaneously in a depth-first, leftmost-first manner, arguing by case analysis on the inference rules that lead to completed evaluation judgments.

Case 1: Rules other than ABS or APP — Equation 7.1 shows that we start with a consistent set of environments and an initial store. For each sub-expression that has been evaluated in sequence, the Soundness Theorem 3.16 guarantees that its value v_i present in the partial execution tree would be consistent with the corresponding type τ_i present in the type derivation tree. Furthermore, we can construct a chain of extensions to the initial store typing, S_i extending S_{i-1} extending $\cdots S_0$, each of which would be consistent with the corresponding store $s_i, s_{i-1}, \ldots, s_0$. If any of these intermediate values entered the dynamic environment through a `let`-binding, then the static environment E_i and the dynamic environment e_i so obtained would also be consistent by construction. Therefore, for each sub-expression evaluation judgment we have,

$$S_i \models v_i : \tau_i \qquad S_i \models e_i : E_i \qquad \models s_i : S_i \tag{7.2}$$

Case 2: ABS Rule — Here, we simply clip the type derivation tree at the abstraction typing judgment in order to emulate the structure of the partial execution tree which produces a function closure immediately. The type-correctness of the function body ensures a consistent static type for the closure by definition of $\models$ (Definition 3.12).

Case 3: APP Rule — This is the interesting case of type reconstruction. By induction hypothesis, the function and the argument expressions evaluate to a closure and a value respectively that are consistent with their types present in the type derivation tree. Furthermore, suppose the base function f present within the closure[11] has arity k with formal parameters $x_1 \cdots x_k$. We need to consider two cases—partial application of the closure to one more argument, and the final application of the closure that generates a new activation frame.

If the current application is a partial application of a closure $\langle \mathsf{clsr}\ f^i, x_{k-i+1}, a_f, e_f^{k-i}\rangle$ to the value v_{k-i+1}, then it immediately produces another closure $\langle \mathsf{clsr}\ f^{i-1}, x_{k-i+2}, a_f, e_f^{k-i+1}\rangle$, where $e_f^{k-i+1} = e_f^{k-i} + \{x_{k-i+1} \mapsto v_{k-i+1}\}$. The type consistency of this value with respect to the result closure type recorded in the type derivation tree follows directly from induction hypothesis. The important point to note is that if some type-variable in the resulting closure type was not being conserved at this application site, then its exact type-hint would also have been supplied at this application site and stored within the closure environment e_f^{k-i+1}.

Now suppose the function has already undergone $k-1$ partial applications before this application to produce a function closure $\langle \mathsf{clsr}\ f^1, x_k, a_f, e_f^{k-1}\rangle$. Therefore, the dynamic

[11] The simple expression language of Chapter 3 does not deal with multi-arity functions directly. Therefore, we assume that each multi-arity function f with arity k in the user program gives rise to a set of functions $f^k, f^{k-1}, \ldots, f^1$ that represent partially applied closures of f accumulating one argument at a time within their environments $e_f^0 \cdots e_f^{k-1}$. The superscript i on the function f^i denotes how many more arguments are needed before the evaluation of the body of the function f is initiated. Likewise, the superscript j on the environment e^j denotes the number of arguments it has accumulated.

APP rule in the caller's partial execution tree looks like,

$$\frac{\begin{array}{c} e_i \vdash a_1/s_i \Rightarrow \langle \mathsf{clsr}\ f^1, x_k, a_f, e_f^{k-1}\rangle/s_{i+1} \\ e_i \vdash a_2/s_{i+1} \Rightarrow v_k/s_{i+2} \\ e_f^{k-1} + \{x_k \mapsto v_k, f \mapsto \langle \mathsf{clsr}\ f^k, x_1, a_f, e_f^0\rangle\} \vdash a_f/s_{i+2} \Rightarrow \_/\_ \end{array}}{e_i \vdash (a_1\ a_2)/s_i \Rightarrow \_/\_}$$

While the static APP rule in the caller's type derivation tree constructed so far looks like,

$$\frac{E_i \vdash a_1 : \tau_k \rightarrow \tau_{k+1} \qquad E_i \vdash a_2 : \tau_k}{E_i \vdash a_1\ a_2 : \tau_{k+1}}$$

We wish to construct an appropriate type derivation sub-tree that models the evaluation of the callee's body.

From the induction hypothesis on the first two clauses and the Soundness Theorem 3.16, we obtain new store typings S_{i+1} and S_{i+2} such that,

$$S_{i+1} \models \langle \mathsf{clsr}\ f^1, x_k, a_f, e_f^{k-1}\rangle : \tau_k \rightarrow \tau_{k+1} \quad S_{i+1} \text{ extends } S_i \quad \models s_{i+1} : S_{i+1} \tag{7.3}$$

$$S_{i+2} \models v_k : \tau_k \quad S_{i+2} \text{ extends } S_{i+1} \quad \models s_{i+2} : S_{i+2} \tag{7.4}$$

Looking at the definition of $\models$ (Definition 3.12), the first clause of Equation 7.3 guarantees that there exists a suitable type environment E_f^{k-1} that is consistent with the closure environment e_f^{k-1} and provides a proper typing for the function body. That is,

$$S_{i+1} \models e_f^{k-1} : E_f^{k-1} \tag{7.5}$$

$$\text{and} \quad E_f^{k-1} \vdash (f^1 \texttt{ where } f(x_1 \cdots x_k) = a_0) : \tau_k \rightarrow \tau_{k+1}$$

$$\Longrightarrow \quad E_f^{k-1} + \{x_k \mapsto \tau_k\} \vdash a_0 : \tau_{k+1} \tag{7.6}$$

The job of the reconstruction algorithm is to construct the type environment E_f^{k-1} and hence build the exact type derivation tree of the function body as given by Equation 7.6.

At compile-time, the static type-map of the function TM_f has already recorded the static type of all the parameters and free identifiers of the function f (Definition 6.1). The reconstruction algorithm simply needs to instantiate this compile-time type environment E_f^{static} to compute the actual type environment E_f^{k-1} as discussed in Section 7.4.4 above. In particular, the algorithm uses the exact type τ_k of the final argument x_k from the application site as well as type-hints contained within the closure environment e_f^{k-1} that allow it to compute the exact types of all the non-conserved type-variables in the type-map TM_f. This completely instantiates the types of all the accumulated arguments $x_1 : \tau_1, \ldots, x_{k-1} : \tau_{k-1}$ and the free identifiers contained within the closure environment e_f^{k-1}.

Having constructed the type environment E_f^{k-1} as above, we can now instantiate the type derivation tree of the body a_f as shown in Equation 7.6. Now it remains to be shown that this type derivation tree is consistent with the evaluation tree of the function body a_f.

We have the following environments,

$$e_f^k = e_f^{k-1} + \{x_k \mapsto v_k\} \qquad E_f^k = E_f^{k-1} + \{x_k \mapsto \tau_k\}$$

Note that all argument and free identifier values contained within the closure environment e_f^{k-1} must be consistent with the type present within the instantiated type environment E_f^{k-1} under the store typing S_{i+1}, *i.e.*, the constructed environment E_f^{k-1} satisfies Equation 7.5. This is because these values have been computed in the earlier part of the evaluation tree which we have already type reconstructed and verified for consistency (Equation 7.2). Since the current store typing S_{i+2} extends S_{i+1}, we have $S_{i+2} \models e_f^{k-1} : E_f^{k-1}$ which is combined with Equation 7.3 and Equation 7.4 to give $S_{i+2} \models e_f^k : E_f^k$. Together with Equation 7.4 and Equation 7.6, we obtain *via* the Soundness Theorem 3.16 that the evaluation of the function body a_f will be consistent with its type elaboration.

Thus, we have successfully reconstructed a consistent type derivation tree shown in Equation 7.6 for the expansion of the partial execution tree due to an arity-satisfied function application within the current frame.

$\square$

Chapter 8

Application Study: Tagless Garbage Collection

In this chapter we study an important application of type reconstruction: Tagless Garbage Collection. We describe the compile-time and run-time support needed to perform garbage collection for a polymorphic language without any type-tags. We have implemented our scheme for the Id language running on a simulator for the *T multi-processor architecture. We describe this implementation and compare its performance with two other storage management schemes: first, a conservative garbage collector that does not use any type information, and second, a compiler-directed storage reclamation scheme that explicitly deallocates objects based on static life-time analysis.

8.1 Introduction

Dynamic memory management is an integral component of modern programming languages such as C, Common Lisp, Standard ML, and Haskell that support the notion of a globally shared heap of objects. It is possible to manage the heap memory manually by means of explicit allocation and deallocation calls, though manual storage reclamation is often a difficult and error-prone process. Usually, it is more convenient to use some automatic mechanism for storage reclamation such as an independent garbage collector that reclaims storage periodically once it is no longer in use.

Traditionally, run-time systems geared towards automatic garbage collection use a tagged object representation model [App90, Wil92]. This enables the garbage collector to distinguish between scalar objects and pointers to heap objects without any support from the user or the compiler, although the user application has to pay the price of tagging and boxing objects and performing continuous tag maintenance.

Recently, storage reclamation techniques with an untagged object representation model have received much attention. The motivation comes from a desire to use the full pointer addressability and native representation for scalars rather than a tagged representation, and to avoid the overhead of continuous tag maintenance. Some techniques, such as conservative garbage collection [Bar88, BW88] and compiler-directed storage reclamation [HJ92, Hic93], do not use any run-time type information. While, garbage collection based on type reconstruction [App89, Gol91, GG92] or explicit type propagation [Tol94] use source type information for identifying and traversing live heap objects. In this chapter, we will study and compare the performance of some of these techniques with a scheme based on full run-time type reconstruction.

8.1.1 Storage Reclamation without Run-time Type Information

In an untagged run-time system, no explicit type information is available at run-time in order to identify and traverse live objects. Still, it is possible to perform garbage collection using a conservative object identification strategy as shown by Boehm and Weiser [BW88]. In this scheme, the garbage collector guesses whether a given value is a scalar or a pointer to a heap object. Typically, the guess is based on certain assumptions about the location and alignment of actual pointer data. Since the guess is conservative, the garbage collector may assume some objects to be live when they are dead and fail to collect them. It may also be possible to compact or copy part of the live data that is definitely known to reside on the heap as shown by Bartlett [Bar88]. The feasibility and efficiency of such schemes depend crucially on the object representation convention used within the run-time system and the possibility of obscuring pointer/non-pointer information within the source language and the compiler.

In another scheme proposed by Hicks [HJ92, Hic93], the compiler performs life-time analysis of objects and automatically inserts explicit deallocation calls for an object that is determined to be dead at a particular point in the program. The compile-time cost of this analysis is substantial since the proposed scheme performs abstract interpretation over the entire program in order to determine the reference patterns of dynamically allocated objects and to approximate their life-times statically. Although, once an object has been determined to be garbage, the run-time cost of deallocating it at an appropriate program point is minimal. Since static analysis is necessarily approximate due to undetermined control flow and sharing or aliasing of objects, this technique is also unable to reclaim all the garbage generated within the program.

8.1.2 Garbage Collection using Run-time Type Reconstruction

The primary motivation for a type-reconstruction-based garbage collection scheme is to take advantage of the enormous compile-time type information available in a statically-typed language in optimizing its run-time performance. In particular, it is possible in such a system to use an untagged and unboxed representation for scalar objects and eliminate type headers for heap objects without compromising the ability to perform complete object identification. All the desired type information may be automatically reconstructed when necessary. Although the cost of type reconstruction may be significant, it needs to be paid only when garbage collection is initiated. Therefore, such a scheme may work very well for scientific applications where numerical performance is of prime concern and garbage collection is expected to happen infrequently and is used in conjunction with explicit storage management. Keeping tagless data also permits easy inter-operability with conventional C and Fortran libraries that do not support tags.

Full run-time type reconstruction also offers some unique advantages that are not present in other schemes for storage reclamation. Having the exact run-time types of objects allows the garbage collector to examine and traverse objects selectively. For example, the collector need not search for heap pointers inside a large array of floating point numbers. Similarly, the scalar fields of a record may be safely skipped. For scientific applications manipulating large numeric arrays, this may constitute a substantial saving in identifying the set of all live objects.

It is also quite easy in this scheme to generate specialized traversal and marking functions for user-defined objects and function activation frames that understand their type and control structure. These functions selectively traverse the fields that point to heap objects as determined by their types, and mark those objects as live. Since these functions are specialized to the type of a particular object, they may be more efficient than interpreting the run-time reconstructed types of the objects.

8.1.3 Related Work

Goldberg and Gloger used type reconstruction to garbage collect a polymorphic language [GG92]. But their system did not guarantee complete type reconstruction. In a situation where a polymorphic function accessed only part of a complex object (see Section 6.1.2), say the spine of a linked list, their system could not determine the full type of the object and therefore could not traverse it completely. The authors argued that the inaccessible parts of the object were garbage anyway and therefore need not be marked as live. Unfortunately, the object could have shared references from other sources that access it farther than the first reference. To deal with such cases, the authors proposed maintaining hash tables of partially traversed data-structures as a way of identifying the extent to which an object was live and therefore should not be garbage collected. This scheme was both cumbersome and costly. On the other hand, our scheme of full type reconstruction allows the garbage collector to traverse the whole object the very first time without using any additional data-structures.

Another interesting scheme has been proposed by Tolmach [Tol94] where type instantiation and propagation is made explicit in the program by converting it into an intermediate form based on the second-order λ-calculus [Rey74, HM93]. Under this transformation, every polymorphic object is parameterized with explicit type parameters for each of its polymorphic type-variables that are instantiated at the time of application to actual type arguments. This explicit run-time type information is used during garbage collection in much the same way as in our scheme. A minor problem in using this scheme is that in order to preserve the call-by-value semantics of ML-like programs, the polymorphic objects appearing on the right-hand-side of a `let`-binding must be restricted to syntactic *values*, *i.e.*, identifiers, constants, or λ-expressions. Wright showed [Wri93] that this restriction is not too serious in practice.

The explicit type parameters used in Tolmach's system are similar in spirit to the explicit type-hints of our type reconstruction scheme, although we add explicit type parameters only for non-conserved type-variables. Our scheme can be considered as an optimal trade-off point between Goldberg's scheme where no explicit type information is propagated at run-time, and Tolmach's scheme where all polymorphic type-variables are instantiated using explicit run-time parameters. We insert explicit type parameters only where necessary assuming that the cost of reconstructing the remaining information at run-time is small.

8.1.4 Goals and Scope of the Study

The main goal of this study is to establish the feasibility of a type-reconstruction based tagless garbage collection scheme (TRGC) and to compare its performance with a conservative garbage collection scheme (CGC) and a compiler-directed storage reclamation scheme (CDSR) that does explicit deallocation.

In order to make a reasonable performance comparison, we have implemented all the three schemes for the same source language, compiler, and the target architecture. Our source language is Id, which is a polymorphic, strongly-typed, implicitly parallel programming language [Nik91]. We are compiling Id for the *T multiprocessor architecture [NPA92, PBGB93] and executing it on an emulator for that machine.

We have chosen a very simple "mark-and-sweep" garbage collection algorithm so that the cost of object identification can be clearly identified during the mark phase. The wall clock performance of the garbage collection algorithm is not our major concern, we are primarily interested in the relative cost of type reconstruction and marking *vs.* the cost of conservative marking. Explicit allocation/deallocation scheme serves as a calibration point representing the essential cost of managing the storage.

8.1.5 Outline

The outline of the rest of the chapter is as follows. Section 8.2 describes the object representation model in Id and summarizes the overall strategy for mark-and-sweep garbage collection based on run-time type reconstruction. Section 8.3 describes the compiler support required. In Section 8.4, we describe the run-time object marking schema based on complete type reconstruction. In Section 8.5, we briefly describe the *T multi-threaded architecture and our implementation of the various storage management schemes on it. Section 8.6 discusses our benchmarks and presents the performance results. Finally, Section 8.7 presents the conclusions.

8.2 Framework for Tagless Garbage Collection

8.2.1 Object Representations and the Memory Model

The Kernel Id intermediate language as shown in Figure 7.1 is an abstract intermediate form that does not take a position on the underlying representation of objects. However, a concrete implementation of a language must specify a representation of objects, which to a large extent, determines its run-time performance and the garbage collection strategy. In this section, we describe the concrete representation of Id objects for our current implementation.

The object representation used in the Id run-time system is independent of the target architecture and only relies upon the assumption of a logically flat, shared, global address space. In order to keep the representation simple and efficient we avoid making any assumptions about boxing and explicit tagging of objects as much as possible. The only assumption necessary to support polymorphism is that we use the same basic unit of memory for all scalar objects and pointers to heap objects which in our case is a single 64-bit word.

Examples of various Id object representations appear in Figure 8.1. Scalar objects are by definition untagged and unboxed in Id. n-dimensional arrays are linearized in row-major order into a flat data-structure that also keeps the bounds in each dimension $(l_1, u_1), \ldots, (l_n, u_n)$ and a set of linearization constants $c_0, \ldots, c_{n-1}$ that are used to compute the linear offset into the array given a n-dimensional index. For an algebraic datatype, depending on the total number m and the arity k_m of its various disjuncts, we may choose one of *product*, *enumerated*, *implicit*, or *explicit* representation. In all cases except when there are more than one non-nullary disjuncts present, we are able to choose an unboxed and untagged representation for the datatype. In particular, when there is exactly one non-nullary disjunct present, as in the case of the *list* datatype, we assume that heap pointers can be distinguished from a small fixed range of integers (say, 0-255), sufficient to represent all the nullary disjuncts of the datatype and no explicit tag is necessary. For some applications, this may save a lot of space and time.

There are two more kinds of objects that are created and manipulated indirectly at run-time by Id programs. These are *function closures* and *activation frames*. In an implementation without lambda-lifting and currying, function closures keep the values of the free identifiers of a function obtained from its lexical environment. In our implementation, all functions are already lambda-lifted, so the closures carry just the curried arguments accumulated under

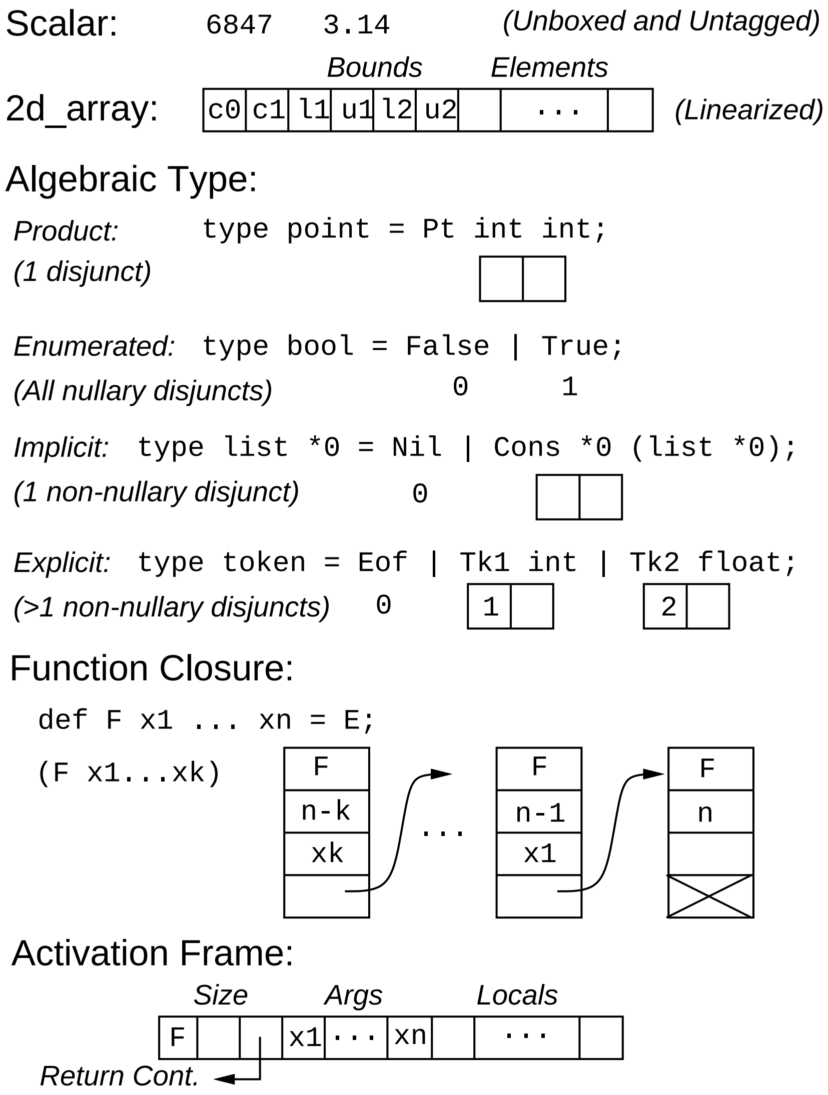

Figure 8.1: Run-time Object Representations for Id.

partial applications. We use the structure depicted in Figure 8.1 which permits sharing of intermediate closures.

An activation frame is a temporary storage area used by an executing function as a scratch pad keeping its input arguments and temporary intermediate values. In Kernel Id, the bound variables of a function constitute the intermediate values that need to be kept within its activation frame for future use.[1] The frame also keeps the return continuation, consisting of the caller's activation frame and the return instruction pointer. In a sequential system, activation frames are usually allocated on a stack. In our parallel execution model, the linear stack of activation frames generalizes to a tree and is managed explicitly by the run-time system.

[1]An intelligent compiler back-end may be able to share some frame slots based on live-variable analysis, but we are ignoring that issue here for simplicity.

8.2.2 Overall Strategy

The overall strategy for a simple mark-and-sweep garbage collection based on run-time type reconstruction is summarized below and described in the following sections:

1. At compile-time, we ensure that every object manipulated by the user program (including function closures and activation frames) is assigned a static, possibly polymorphic, datatype that accurately describes the structure of that object (Section 8.3).
2. When the garbage collector is invoked at run-time, first we reconstruct the type of every activation frame present within the current dynamic call tree using the algorithm described in the last chapter. The reconstruction mechanism instantiates the compile-time type description of each activation frame to its exact run-time type.
3. Next, within the mark phase of the garbage collector, each slot of a reconstructed frame is examined and its reconstructed type is used to mark the heap objects reachable from that slot as live. This may be done in two ways: the reconstructed types may be directly interpreted to identify and traverse the heap objects, or the compiler may automatically generate specialized traversal and mark routines that are appropriately composed at run-time in order to mark the live objects (Section 8.4).
4. Finally, the unmarked heap objects are reclaimed as garbage by sweeping the entire heap.

8.3 Compiler Support for Object Identification

8.3.1 Visible and Invisible Datatypes

The scalar basetypes, algebraic datatypes, and array types in Kernel Id correspond to pure data-objects whose types are directly *visible* at the source language level. There is a direct, fixed mapping from the source types of these objects to their internal representations as described in Section 8.2.1. This mapping may be directly used in traversing these objects at run-time once their exact source type is determined.

On the other hand, arrow types ($\rightarrow$) correspond to two different run-time objects: function closures which behave like data-objects that must be garbage collected, and activation frames which are control-objects consisting of the live object root set. Neither of these is modeled completely by the source-level arrow type. This is because the visible type signature of a function does not provide any clue regarding the types of the arguments hidden inside its closure, nor does it provide any information about the local variables kept within the function's activation frame. In order to treat all Id run-time objects uniformly in terms of Id source types, we define *invisible* source-level datatypes for function closures and activation frames that provide an exact description of their contents.

8.3.2 Modeling Function Closures

In order to simplify the type reconstruction analysis, we model the closures corresponding to partial applications of a function as disjuncts of an invisible algebraic datatype that is automatically derived at compile-time from the corresponding function signature. This derivation is shown in Figure 8.2. The various disjuncts of this hidden datatype represent successive partial applications of the function and identify the number and the types of the accumulated arguments. This indirect model captures all the necessary type information required to traverse the actual run-time representation of a function closure as shown in Figure 8.1. Given a run-time closure object, we can map it to an algebraic disjunct in this model by examining its function code-block pointer and the remaining arity slot. Then, given the exact algebraic type of the closure, the arguments contained within the closure can be traversed using the argument types of the mapped disjunct.

As an example, below we show a function `eqlen` that compares the length of two lists. We also show its Hindley/Milner visible source type and its automatically derived hidden closure datatype:

Example 8.1:

```
def eqlen l1 l2 =              % eqlen :: ∀αβ.(list α) → (list β) → bool
  { len1 = length l1;
    len2 = length l2;
    p = len1 == len2;
   in p };

type eqlen_closure α β =       % Hidden Closure Type
  eqlen_ap0
| eqlen_ap1 (list α);

f = eqlen (1:nil);             % f :: ∀β.(list β) → bool
                               % f :: ∀β.(eqlen_closure int β)
```

The constructor $\texttt{eqlen\_ap}_0$ models the closure representation of the `eqlen` function itself, while $\texttt{eqlen\_ap}_1$ represents the closure formed by a partial application of the `eqlen` function to one argument. The example also shows the source type and the invisible type of a partial application of the `eqlen` function.[2] Note that the invisible type records the fact that the hidden first argument within the closure is a list of integers while this information is not present in the source type.

There is no need to make a closure for `eqlen` with two arguments since at that point its arity is fully satisfied and the application gives rise to an activation frame instead of a function closure.[3] Finally, note that the invisible closure datatype is parameterized by *all* the type-variables present in the source type of the function. This is necessary in order to model the exact run-time types of all the arguments contained within the closure.

8.3.3 Modeling Activation Frames

Function activation frames are modeled using an automatically derived, invisible datatype called the function *framemap* as shown in Figure 8.2. This is simply a record datatype with a field for every actual frame-slot (*c.f.* Figure 8.1). Besides the scalar datatype fields for the code-block entry point, the frame size and the return continuation, the framemap record the types of the function arguments and the local identifiers used within the function body.

Abstractly, the framemap of a function provides a logical subset of the type information recorded within its type-map (Definition 6.1) and is parameterized by the same type-variables. The framemap simply provides a concrete static image of a function's dynamic activation frame and therefore may depend on its actual implementation on a given platform. After type reconstruction is complete, each activation frame is associated with a fully instantiated type-map from which an appropriate framemap instance can be derived in order to traverse the heap objects accessible through each frame-slot.[4]

---

[2] ":" is the infix *cons* constructor for lists.

[3] However, under delayed or lazy evaluation, we may need to keep track of such *thunks*.

[4] In our current implementation, the type-map produced by the Id compiler is tailored to the structure of

INVISIBLE DATATYPES

Given a Function Declaration: $\texttt{def}\ F\ x_1 \cdots x_n = E$

$F :: \forall \alpha_1 \cdots \alpha_m . \tau_1 \rightarrow \cdots \rightarrow \tau_n \rightarrow \tau_{n+1}$

Let $(z_1 :: \sigma_1) \cdots (z_m :: \sigma_m)$ be the locally bound identifiers of E.

1. Define Function Closure Datatype:

$$\begin{array}{lll} \texttt{type}\ F\texttt{\_closure}\ \alpha_1 \cdots \alpha_m = & & F\texttt{\_ap}_0 \\ & | & F\texttt{\_ap}_1\ \tau_1 \\ & | & \cdots \\ & | & F\texttt{\_ap}_{n-1}\ \tau_1 \cdots \tau_{n-1}; \end{array}$$

2. Define Function Framemap Datatype:

$$\begin{array}{lll} \multicolumn{3}{l}{\texttt{type}\ F\texttt{\_framemap}\ \alpha_1 \cdots \alpha_m =} \\ \{\texttt{record} & (F :: \mathit{code}) & \text{% Code-Block Entry Point} \\ & (N :: \mathit{int}) & \text{% Frame Size} \\ & (R :: \mathit{cont}) & \text{% Return Continuation} \\ & (x_1 :: \tau_1) & \text{% Arguments} \\ & \cdots & \\ & (x_n :: \tau_n) & \\ & (z_1 :: \sigma_1) & \text{% Local Identifiers} \\ & \cdots & \\ & (z_m :: \sigma_m)\ \}; & \end{array}$$

Figure 8.2: Automatic Derivation of Invisible Datatypes.

As an example, we show the framemap datatype for the `eqlen` function given above:

Example 8.2:

```
type eqlen_typemap α β =
  {record
     (eqlen   :: code)
     (size    :: int)
     (retcont :: cont)
     (l1      :: (list α))
     (l2      :: (list β))
     (len1    :: int)
     (len2    :: int)
     (p       :: bool) };
```

8.3.4 Run-time Type Encodings

Run-time type reconstruction requires an encoding of all the visible and invisible datatypes of a program that is used to encode type-hints and to represent the exact run-time types of objects during type reconstruction. We showed such an encoding and decoding scheme in Figure 7.3 in Chapter 7. In this scheme, each algebraic datatype T^n is encoded into a corresponding static

the activation frames used in the *T run-time system. Therefore, we directly use the type-map of a function to traverse its activation frame.

type descriptor $\overline{T^n}$ that contains all the necessary compiler information about its arity, internal field structure, and its representation.

Our compiler generates static type descriptors for all the user-defined algebraic datatypes and the automatically derived closure and framemap datatypes (Figure 8.2) for each declared function within the program. These static descriptors are linked together with the object program and are used by the run-time system during type reconstruction. Run-time types are encoded as a flat array of static type descriptors using back-pointers to preserve sharing. This representation permits very efficient copying, unification, and instantiation operations on encoded types. The packing and unpacking of these encoded types is carried out on the fly within the run-time system.

8.4 Run-time Object Traversal and Marking

In this section, we describe our scheme for object traversal and marking assuming complete type reconstruction has been performed. We present two mechanisms:

Interpreted Marking – In this mechanism, the encoded types generated by type reconstruction are directly used to guide the traversal and marking of the heap objects.

Compiled Marking – In this mechanism, the compiler automatically generates marking functions for each datatype in the program based solely on the static type information. These functions are appropriately composed at run-time using the reconstructed types and then directly applied to the corresponding objects.

Both mechanisms are specified as a set of mark functions, one for each basetype, array type, and algebraic datatype present in the program. The algebraic datatype could be a user-defined datatype (Figure 7.1) or an invisible datatype defined by the compiler for function closures and activation frames (Figure 8.2).

8.4.1 Interpreted Marking

The *Interpreted Marking Schema* $\mathcal{M}$ for a type T^n is shown in Figure 8.3. In this schema, for each type T^n with n type parameters $\alpha_1 \cdots \alpha_n$, we define a mark function `mark_T` that is parameterized by n corresponding encoded type arguments $z_1 \cdots z_n$. At run-time, this function is supplied with the exact encoded type instantiation of its type parameters, say $\overline{\tau_1} \cdots \overline{\tau_n}$, which produces an appropriate marking function for an object with type $(T^n\ \tau_1 \cdots \tau_n)$.

The internal structure of the mark functions closely follows the structure of their corresponding datatypes. The polymorphic, bound type-variables of a type-scheme are mapped to dummy mark functions because polymorphic objects contain no information. Similarly, all our base types are scalars, so the mark functions for them do nothing. The mark function for arrays and algebraic datatypes first mark the object itself and then proceed to mark their internal components. This is achieved by first computing the exact run-time type encoding for each of the components and then interpreting that encoding. The code to compute the exact type encoding is directly compiled into the mark functions using the **TEnc**$[\![\,]\!]$ scheme shown earlier (Figure 7.3).

The overall process of interpretive marking is governed by the top-level type-code interpretation function shown in Figure 8.4. Here, we have generalized the type-code interpretation scheme **Interpret**$[\![\,]\!]$ for an arbitrary datatype schema $\mathcal{R}$ such as the marking schema $\mathcal{M}$ of Figure 8.3. This process unpacks the encoded type and invokes the schema function for the appropriate type descriptor passing it the rest of the encoded type arguments. In the present

MARKING SCHEMA $\mathcal{M}$

Given a polymorphic type-variable α_i, define $\mathcal{M}[\![T^0_{\alpha_i}]\!] = \texttt{mark\_}T_{\alpha_i}$, where
$\texttt{def mark\_}T_{\alpha_i}\ () = \lambda x.()$

Given a Type T^n, define $\mathcal{M}[\![T^n]\!] = \texttt{mark\_}T$, where

1. T^0 is a BaseType ($int \mid float$):
   $\texttt{def mark\_}T\ () = \lambda x.()$

2. T^1 is an ArrayType ($\texttt{n}d\_array\ \alpha$):
   $\texttt{def mark\_n}d\_array\ (z) =$
   $\lambda a.\{\ Mark(a);$
   $\quad (l_1, u_1), \ldots, (l_n, u_n) = bounds(a);$
   $\quad \texttt{for}\ i_1 \leftarrow l_1\ \texttt{to}\ u_1\ \texttt{do}$
   $\quad \ldots$
   $\quad\quad \texttt{for}\ i_n \leftarrow l_n\ \texttt{to}\ u_n\ \texttt{do}$
   $\quad\quad\quad \textbf{Interpret}[\![\mathcal{M}]\!]\ (\textbf{TEnc}[\![\alpha]\!]\ \{\alpha \mapsto z\})\ a[i_1, \ldots, i_n];$
   $\quad \}$

3. T^n is an Algebraic DataType ($T^n\ \alpha_1 \cdots \alpha_n$):
   $\texttt{def mark\_}T\ (z_1, \ldots, z_n) =$
   $\lambda x.\{\ Mark(x);$
   $\quad \texttt{Case\_}T\ x\ \texttt{of}$
   $\quad\quad C_1\ x_1 \cdots x_{k_1} = \{\ \textbf{Interpret}[\![\mathcal{M}]\!]\ (\textbf{TEnc}[\![\tau_{11}]\!]\ \{\alpha_i \mapsto z_i\})\ x_1;$
   $\quad\quad\quad \ldots$
   $\quad\quad\quad \textbf{Interpret}[\![\mathcal{M}]\!]\ (\textbf{TEnc}[\![\tau_{1k_1}]\!]\ \{\alpha_i \mapsto z_i\})\ x_{k_1};\ \}$
   $\quad \mid\ \ldots$
   $\quad \mid\ C_m\ x_1 \cdots x_{k_m} = \{\ \textbf{Interpret}[\![\mathcal{M}]\!]\ (\textbf{TEnc}[\![\tau_{m1}]\!]\ \{\alpha_i \mapsto z_i\})\ x_1;$
   $\quad\quad\quad \ldots$
   $\quad\quad\quad \textbf{Interpret}[\![\mathcal{M}]\!]\ (\textbf{TEnc}[\![\tau_{mk_m}]\!]\ \{\alpha_i \mapsto z_i\})\ x_{k_m};\ \}$
   $\quad \}$

Figure 8.3: Generating Mark Functions for Datatypes.

Given a Datatype Schema $\mathcal{R}$, define

$$\textbf{Interpret}[\![\mathcal{R}]\!]\ \overline{\tau} = \begin{array}{l} \{\ \texttt{Case}\ head(\overline{\tau})\ \texttt{of} \\ \qquad \overline{T_1^n} = (\mathcal{R}[\![T_1^n]\!])\ args^n(\overline{\tau}) \\ \quad \mid\ \overline{T_2^m} = (\mathcal{R}[\![T_2^m]\!])\ args^m(\overline{\tau}) \\ \quad \mid\ \cdots\} \end{array}$$

Figure 8.4: Type-code Interpretation at Run-time.

Given a Datatype Schema $\mathcal{R}$ and a Translation Environment Γ_R, define

$$\begin{array}{lcl} \mathbf{Compile}[\![\mathcal{R}]\!]\ \Gamma_R \quad \alpha & = & \Gamma_R(\alpha) \\ \mathbf{Compile}[\![\mathcal{R}]\!]\ \Gamma_R \quad (T^n\ \tau_1 \cdots \tau_n) & = & (\mathcal{R}[\![T^n]\!])\ (\mathbf{Compile}[\![\mathcal{R}]\!]\ \Gamma_R\ \tau_1, \ldots, \mathbf{Compile}[\![\mathcal{R}]\!]\ \Gamma_R\ \tau_n) \\ \mathbf{Compile}[\![\mathcal{R}]\!]\ \Gamma_R \quad \forall \alpha_1 \cdots \alpha_n.\tau & = & \mathbf{Compile}[\![\mathcal{R}]\!]\ \Gamma_R\ (\tau[T^0_{\alpha_i}/\alpha_i]) \end{array}$$

Figure 8.5: Type-based Translation at Compile-time.

case, ($\mathbf{Interpret}[\![\mathcal{M}]\!]\ \overline{\tau}\ x$) traverses and marks the object x according to its exact run-time type encoding $\overline{\tau}$ by recursively instantiating and invoking the mark functions associated with the type descriptors in τ. Other structured datatype schema such as a printing schema or an I/O schema may also be defined and interpreted in a similar manner.

In our current implementation, the type-code interpretation mechanism of Figure 8.4 is built into the run-time system. The marking process is invoked for each type-reconstructed activation frame present in the dynamic activation tree. The run-time system constructs the exact run-time type encoding of every frame-slot in the given activation frame and then directly dispatches to the appropriate marking function based on the datatype class as specified in Figure 8.3. The marking process is further optimized based on the actual representation chosen for a particular class of datatypes as shown in Figure 8.1. For example, the marking function for linearized arrays computes the total size of the array and marks each of its elements in a single loop. In case of algebraic types, nullary disjuncts under enumerated or implicit representation are never marked, a product disjunct is always marked, and a tag dispatch is made for explicitly tagged disjuncts. Finally, the hidden arguments inside function closures are traversed and marked according to their reconstructed hidden closure types.

8.4.2 Compiled Marking

Rather than interpreting type encodings as in the interpreted marking schema, it is also possible to generate compiled marking functions for each datatype that know how to traverse the object directly without any type interpretation. In this *Compiled Marking Schema* $\mathcal{M}'$, for each datatype T^n the compiler automatically generates a mark function `mark'_T` that is parameterized by n mark function arguments $f_1 \cdots f_n$ instead of encoded type arguments. This alternate marking schema $\mathcal{M}'$ can be directly obtained from our interpreted marking schema $\mathcal{M}$ shown in Figure 8.3 by replacing the recursive call for interpretation:

$$\mathbf{Interpret}[\![\mathcal{M}]\!]\ (\mathbf{TEnc}[\![\tau]\!]\ \{\alpha_i \mapsto z_i\})$$

by a type-based function composition:

$$(\mathbf{Compile}[\![\mathcal{M}']\!]\ \{\alpha_i \mapsto f_i\}\ \tau)$$

This transformation expresses the fact that building the exact run-time type encoding of an object and then interpreting it to guide the traversal and marking is functionally equivalent to directly traversing it using a compiled marking function that knows the structure of that object.

The general mechanism of type-based function composition ($\mathbf{Compile}[\![\mathcal{R}]\!]\ \Gamma_R\ \tau$) for an arbitrary schema $\mathcal{R}$ (such as the compiled marking schema $\mathcal{M}'$) is shown in Figure 8.5. Generating compile-time type encodings as shown in Figure 7.3 may be thought of as a special case of this mechanism. This mechanism translates a given static type τ into a composition of schema functions specified by $\mathcal{R}$ under a translation environment Γ_R that maps free type variables of τ to schema-dependent values. For the case of compiled marking schema, $(\mathbf{Compile}[\![\mathcal{M}']\!]\ \{\alpha_i \mapsto f_i\}\ \tau)$ creates a function composition that is capable of marking an object whose type is a run-time instance of the static type τ. Note that the marking function so generated does not contain any type-code interpretation. Its execution directly results into the appropriate traversal and marking of the given object.

The compiled marking process is initiated by converting the reconstructed type-map of each activation frame into a composition of compiler-generated marking functions. This translation is similar to the type-based function composition shown in Figure 8.5 except that it operates on type encodings rather than static types. The resulting function composition may be directly applied to the given activation frame to mark all heap objects reachable from that frame. The compiled marking schema is currently unimplemented.

8.4.3 Variations on Marking Schemes

The interpreted and the compiled marking schemes described above are just a few among a full spectrum of possible marking schemes that depend on the degree of type specialization performed at compile-time and degree of type interpretation performed at run-time. For instance, it is possible to have a marking schema that takes an intermediate position between the completely interpreted schema $\mathcal{M}$ and the completely compiled schema $\mathcal{M}'$. In this schema, calls to the top-level interpretive dispatch (Figure 8.4) may be statically specialized to call the marking functions of schema $\mathcal{M}$ directly, although dynamic type-hints may still have to be interpreted at run-time.

It is also possible to specialize the type-hint propagation and the type reconstruction mechanism described in the last chapter (Section 7.2 and Section 7.3) for the explicit purpose of object marking. In this scheme, the compiler would insert code to generate and propagate type-hints (Section 7.2.3) that consist of compositions of mark functions rather than run-time type encodings. The type reconstruction algorithm (Section 7.4.4) would also be modified to deal with such type-hints and the algorithm would return a higher-order composition of mark functions for the given activation frame rather than a reconstructed type-map. The mark function so obtained would be directly applied to the activation frame to mark all heap objects accessible from it.[5]

An independent variation for any of the compiled marking schemes is to generate as many specialized marking functions as possible at compile-time for every static type occurring in the program rather than generating compositions of a fixed set of datatype marking functions as shown above. This would clearly reduce the overhead of using higher-order marking functions.

8.5 *T Implementation

*T is a parallel, distributed-memory machine with a high performance interconnection network [NPA92, PBGB93]. The *T architecture extends a basic RISC instruction set with low-overhead, user-mode communication and synchronization primitives. The details of the architecture may be found elsewhere [Bec92]. In this section, we briefly summarize some of the

[5] Readers familiar with Haskell's *type classes* [HWe90, WB89] would immediately recognize that in Haskell, we can accommodate all variations of type reconstruction and its applications by declaring a universal class `trec` that provides type encodings, mark functions, print functions etc. as independent methods.

design features and the terminology of the *T architecture that are relevant to the implementation of Id on *T and then describe our implementation of distributed garbage collection on this machine.

8.5.1 Multi-threaded Execution: Processor View

In our study, we used a simulator for the *T architecture based on Motorola's 88110MP processor. The 88110MP is a super-scalar RISC processor extended with an on-chip message and synchronization unit (MSU) which provides hardware support for scheduling *microthreads*. A microthread is a compiler-defined sequence of instructions executing within the context of an activation frame. A *microthread descriptor* identifying a microthread consists of an instruction pointer (IP) and a frame pointer (FP) (refer Figure 6.2). A microthread, by definition, executes to completion once it has been invoked. It may send messages or fork other microthreads that are deposited in a stack of ready-to-run microthreads.

*T processors communicate with each other by sending *messages via* the network. Messages consist of 4 to 24 32-bit words. Due to the on-chip message unit, *T messages may be dispatched and handled very quickly using the general-purpose processor registers directly (6 and 12 instructions respectively for a full-sized message). Messages always contain a microthread descriptor as the first two words of payload. Normally, messages are handled by invoking the microthread described within the message, so these microthreads are termed *message handlers*.

A microthread's last operation is to schedule the next microthread of the highest priority which is selected from a simple priority queue consisting of handlers of incoming messages, the microthread stack, and several microthread registers. Message handlers have higher priority than computation microthreads.

8.5.2 Multi-threaded Execution: System View

*T runs a Unix-like operating system. A parallel job running on *T consists of a separate process, or a *player*, on each processor. Players belonging to the same parallel job are scheduled at the same time on their respective processors by the operating system. The players have independent 32-bit virtual address spaces, but may refer to a global 64-bit address space through the MSU by sending messages to each other.

The Id compiler and its run-time system for *T provide the high-level abstraction of a single, implicitly parallel program running within a shared, global address space as shown in Figure 6.2. The Id compiler statically partitions the user program into several microthreads that are scheduled dynamically during execution. Microthreads communicate and synchronize with each other *via* messages. Microthreads belonging to a single Id procedure execute within the context of a shared activation frame and may also communicate with each other *via* the frame. Since successively scheduled microthreads on a processor may be completely independent, the general-purpose registers within the processor are kept local to a microthread and are not used to communicate data across microthreads. However, registers may still be used to pass parameters to C functions called within a single microthread.

The Id run-time system consists of the frame manager, the heap manager, and protocol handlers for I-structure and M-structure memory operations [CCF$^+$93]. All run-time system calls are initiated and serviced as *split-phase transactions*. A microthread sends a message to a run-time system request handler passing it the descriptor of a microthread that would receive the reply. The request handler services the request and returns the result in a message to the reply handler provided with the request. This scheme ensures that computation microthreads never block the processor pipeline and can always run to completion.[6] This invariant guarantees that run-time system exceptions such as running out of frame or heap memory always happen at the boundary of a computation microthread. At that moment, none of the general-purpose registers contain any live data and the complete root set of heap objects is available within the tree of activation frames.

The Id run-time system sets up the players participating in a parallel job to continuously execute a microthread dispatch loop where microthreads are scheduled according to the priority scheme described earlier. One of the players (processor 0) is setup to allocate the root activation frame and launch the first microthread along with its user-supplied arguments. It also receives the final result and coordinates the termination of the parallel job.

8.5.3 Memory Organization

For the purpose of executing Id programs, the *T machine is logically divided into two kinds of nodes: computation nodes and memory nodes (see Figure 8.6). The computation nodes manage the dynamic tree of activation frames and execute computation microthreads while the memory nodes manage the heap memory and handle various protocols for memory references.

The address space of a player running on a *T processor is divided into several areas that are themselves distributed or replicated across the nodes as shown in Figure 8.6.

The code and static data areas are replicated on all nodes — each node gets a copy of the whole program and all of its constants. Each node also has a stack that is used for calling into C procedures from Id. The Id run-time system is implemented in C and may also use the C stack.

The frame area on the computation nodes contains the activation frames for every Id procedure invocation. When a procedure is invoked, the run-time system chooses a processor on which to allocate its frame according to a built-in load balancing strategy. Then, the run-time system sends a frame allocation request to that processor in a split-phase transaction, which allocates a frame in its own frame area and returns a pointer to it to the calling routine. This mechanism distributes the dynamic tree of activation frames across all the computation nodes.

An activation frame is deallocated by the last microthread of its associated procedure and may be reused subsequently. In order to avoid confusion due to stale data lying around from previous allocations, the Id compiler arranges the first microthread of each procedure to clear all frame-slots that may contain pointers. This helps in identifying valid data within the frame during garbage collection.

The heap area on the memory nodes contains all of the heap-allocated Id objects. The heap area is further divided into the interleaved and the non-interleaved area. The non-interleaved area is used for small sized objects contained wholly within the same node, while the interleaved area is used to allocate large objects that are spread across all the memory nodes to avoid allocation imbalance and reduce memory contention. In order to simplify our study, we only used the non-interleaved heap area.

In our implementation of Id on *T, all scalar objects and pointers to heap objects are 64 bits in size. Furthermore, these pointers are always aligned on 8-byte boundaries when stored in memory. Each 64-bit double word in the heap has an associated 2-bit presence value in the presence-bit area. These presence bits are used to implement Id's I-structure [ANP89], and M-structure [BNA91] synchronization operations.

[6] If the network is blocked, the message is buffered and is tried again at a later point. Thus, the currently executing microthread is guaranteed to terminate without blocking.

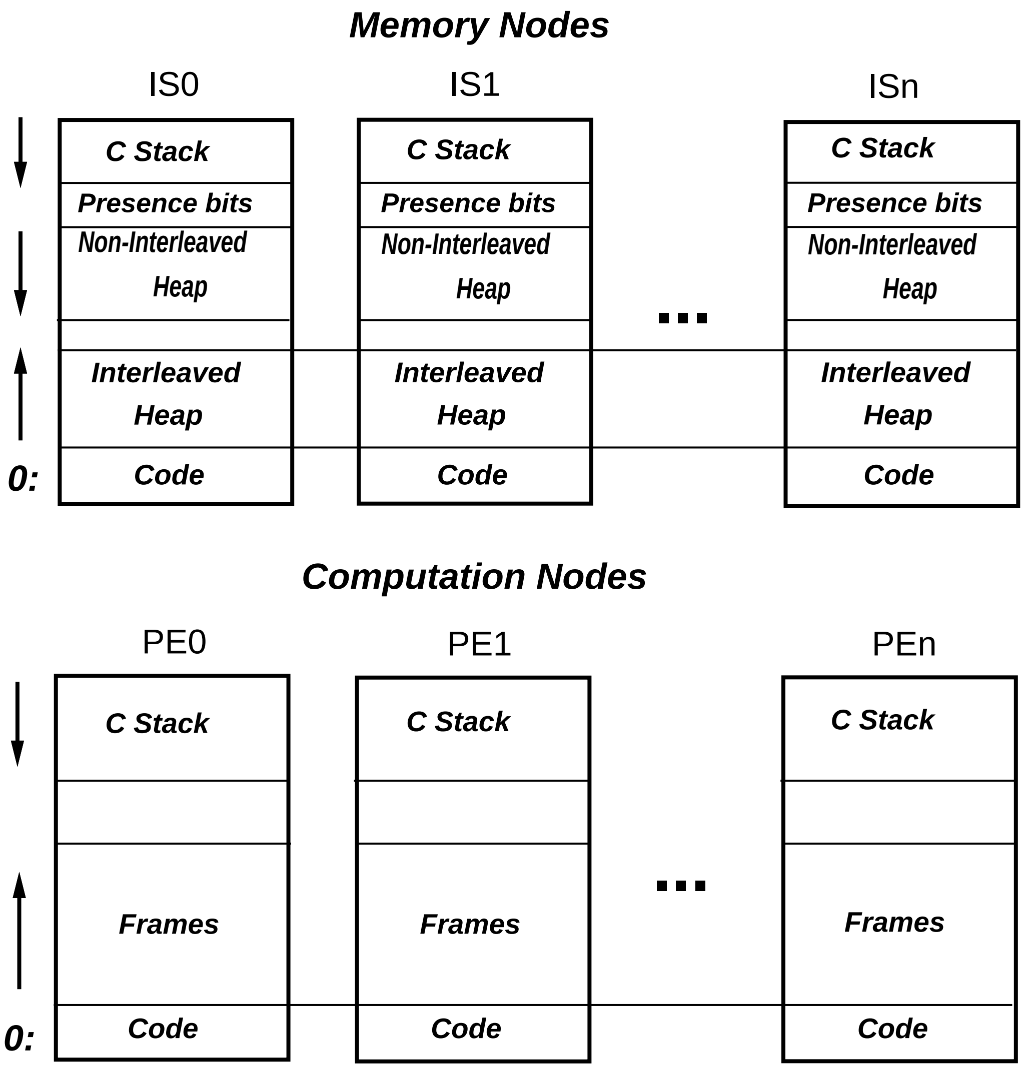

Figure 8.6: The Organization of Computation Nodes and Memory Nodes in the *T machine.

We also use the non-interleaved heap area to keep any deferred-read and locked-take continuations for the I-structure and M-structure operations respectively. These continuations represent incomplete split-phase memory accesses whose second phase would complete when the corresponding heap data becomes available. Therefore, the heap objects carrying these continuations are always considered to be live and should never be garbage collected. On the other hand, since our system does not perform tail-calls, pointers to activation frames contained within such continuations are always accessible through the dynamic tree of activation frames. Therefore, these continuations do not have to be scanned for live pointers. Currently, our run-time system permanently marks such objects as live and manages their allocation and deallocation separately. Also, the garbage collector treats their contents as scalar data. A cleaner solution would have been to designate a separate heap area for allocating such deferred continuations so that the garbage collector never sees them.

Our compiler and run-time system never store a pointer to the interior of an object in a frame-slot or another Id object. Therefore, a pointer found within a frame or a heap object always points to the head of the *active area* of the object. The active area of the object is actually preceded in memory by some information managed by the run-time system including the object's size (used for deallocation), a mark-bit (used by the garbage collector), and the time when it was allocated (in instruction cycles — for statistics collection).

8.5.4 Garbage Collection on *T

Garbage collection on *T can be initiated either by request from the Id program or by the run-time system when one of the processors finds out that it is running out of heap storage. Our current policy is to initiate garbage collection when the allocated storage on a node reaches a specified fraction (say, 0.75) of its total storage.

Since the heap is shared globally, all processors must participate in a global garbage collection. Therefore, when one processor decides to do garbage collection, all other processors are informed about it. Currently, we have implemented a simple stop-and-collect garbage collection scheme.

First, the computation nodes stop processing computation microthreads and drain all messages out of the network because the messages may carry live pointers to heap objects. As messages are drained from the network, their handlers are invoked. Our compiler ensures that the computation message handlers may modify memory locations or fork other microthreads, but they are not allowed to send more messages.[7] We can handle all messages and eventually reach quiescence, as long as we do not run any threads scheduled by the message handlers. Since we invoke message handlers as the network drains, there are no queues of messages to consider as part of the root-set during garbage collection.

Once the network is drained, all processors synchronize and then initiate the mark phase. In this phase, all live and reachable objects residing on the memory nodes are marked according to one of the object identification techniques starting from the distributed tree of activation frames residing on the computation nodes. This process requires global communication among processors to mark objects distributed across the machine. After global marking is completed on all nodes, the processors synchronize again and then each memory node begins a local sweep phase. A final synchronization is performed after sweeping is completed on all nodes, and then the Id threads are allowed to resume computation on the computation nodes.

Type-Reconstructed Garbage Collection

The mark phase of the Type-Reconstructed Garbage Collection (TRGC) follows the compiler-directed object identification scheme described earlier. Currently, we have only implemented the interpreted marking scheme with full type reconstruction as described in Section 8.4.1.

During the mark phase, the frame memory of each computation node is traversed locally to find the activation frames that belong to the current dynamic activation tree. Each activation frame that is currently in use is type-reconstructed according to the algorithm shown in Figure 7.5. Since the dynamic activation tree is distributed across processors, this process may require sending messages to non-local parent activation frames in order to obtain their use-type instantiations.

Once a frame is reconstructed, its slots are searched for heap objects to be marked using their fully reconstructed types. We directly follow the type-code interpretation scheme of Figure 8.4

[7] The run-time system message handlers are still allowed to send reply messages, but the number of such messages is fixed.

by examining the type constructor for the current frame-slot to see if it refers to a structured datatype. If so, the value in the frame-slot is parsed as a pointer and a request for marking the corresponding object is sent to its home node along with its fully reconstructed type packed within the requesting message. At the home node, the object and its contents are marked according to the marking schema shown in Figure 8.3.

Note that, although type reconstruction of a frame must precede marking within that frame, it may be overlapped with type reconstruction or marking of other frames or heap objects.

Conservative Garbage Collection

The mark phase of the Conservative Garbage Collection (CGC) requires no source type information. Conservative garbage collectors use a simple, conservative test to determine whether a value in a frame or a heap object is a pointer to another object. Since pointers are identified conservatively, CGC may assume that there are live references to an object when there are none, therefore some objects may remain uncollected. Also, CGC cannot compact or copy all objects because conservatively identified pointers cannot be updated. However, there are some more sophisticated schemes that allow compaction and/or copying of a large fraction of the heap objects [Bar88]. Finally, CGC has no knowledge of the source types, therefore it must examine every slot of every reachable object and no short-circuiting based on scalar-type information is possible.

As in the case of TRGC, the mark phase of CGC begins on the computation nodes by traversing their frame memory and identifying the activation frames currently in use. For each activation frame in use, we apply the conservative pointer test on each of its frame-slots as follows:

1. First, we check to see if the 64-bit value contained within the frame slot is non-zero and is aligned to a 64-bit boundary. If not, then the value is a scalar.

2. Next, we parse the value as a potential global pointer and determine its home node. If the node address falls outside the known range of addresses for memory nodes, the value is a scalar.

3. Finally, we send a message to the home node to check if the value is a valid pointer. At the home node, we test whether the value points within the allocated heap area and that it points to the head of an actual heap object. The latter test is made possible because the run-time system marks the head of each allocated object with a special presence-bit pattern. Furthermore, the system guarantees that actual pointers never point to the interior of objects. Therefore, this test may be carried out by simply checking for the special presence-bit pattern at the head of the pointer value. If this test succeeds then the value is considered to be an actual pointer and the object is marked, otherwise the value is taken to be a scalar.

The test may mark some objects that are not actually reachable because a value in memory happens to look like a pointer to that object. However, the test is guaranteed to mark only actual heap objects because it checks for the special allocation presence-bit pattern.

Once a value has been determined to be a pointer, the fields of the object it points to are scanned for potential references to other objects in a similar fashion.

Compiler-Directed Storage Reclamation

For comparison purposes, we have also implemented the explicit, compiler-directed storage reclamation scheme (CDSR) within the same compiler and run-time system framework. In this scheme, no separate garbage collection needs to be performed: the compiler inserts code to deallocate an object when it can determine the object to be garbage. This analysis has a substantial compile-time cost compared to the other two storage management schemes. Also, the static analysis may not be able to reclaim all the garbage that is generated by the program.

The run-time costs of this scheme may be divided into a small synchronization cost that schedules the deallocation of an object when all its references are dead and the actual cost of deallocating the object. The former cost is negligible and is also hard to separate from the user program because it is built into the microthread partitioning and synchronization of the user program. The second cost is the same as the basic cost of sweeping the unused objects as in the other garbage collection schemes and therefore forms the basis of our comparison with those schemes.

We use the CDSR scheme to compare its relative storage management efficiency to that of the garbage collected schemes. It is also possible to simultaneously use the explicit storage management scheme to get most of the large objects along with a garbage collector that catches the smaller, harder to analyze objects. We believe that a mixed approach may yield better performance than either scheme on its own.

8.6 Performance Results and Analysis

We are interested in two aspects of the performance of the type-reconstructed garbage collection (TRGC): how long it takes to garbage collect, and how much garbage it reclaims. We compared several programs running with TRGC, conservative garbage collection (CGC), and compiler-directed storage reclamation (CDSR).

In preparing a uniform execution platform, we naturally had to accommodate the requirements of each storage management scheme within the same run-time system. This resulted in a system that was not tuned to any particular storage management scheme. For instance, a copying or compacting garbage collector could not be used for TRGC since our simple-minded scheme for conservative garbage collection would not work in that setup. Similarly, the run-time system had to maintain free-lists for reclaimed objects since we wanted to perform explicit storage management within the same framework.

Thus, the results we obtained cannot be treated as an absolute measure of performance for any particular scheme. On the other hand, they provide a good measure of relative performance of the object identification mechanisms studied and also characterize systems where more than one storage management strategy is used.

8.6.1 Benchmark Runs

We used four different benchmarks. *Quicksort* is the standard recursive algorithm for sorting N list elements parameterized by a polymorphic comparison predicate. *Paraffins* generates and counts the number of distinct paraffin isomers of up to N carbon atoms. *Gamteb* is a Monte Carlo simulation of N photons impinging on a carbon rod divided into two cells. Finally, *Wavefront* consists of 10 iterations of a successive over-relaxation kernel of a $N \times N$ matrix containing floating-point data.

Quicksort				Instruction Cycles (×1000)							
Mode	Input N	Heap (Wds)	GCs	Id	Id RTS					Idle	Total
					Basic	Mark	Sweep	TREC	Total		
TRGC	25	5628	2	488	209	109	17	22	372	3	863
CGC	25	5640	2	488	208	137	16	0	367	5	860
CDSR	25	5236	0	519	193	0	0	0	195	7	721
TRGC	50	8640	2	1121	513	201	30	43	812	8	1942
CGC	50	8628	2	1113	497	179	30	0	714	5	1833
CDSR	50	10936	0	1185	466	0	0	0	469	7	1661
TRGC	75	15000	2	1736	810	492	51	129	1549	7	3293
CGC	75	15004	2	1717	783	179	48	0	1019	7	2743
CDSR	75	17328	0	1852	747	0	0	0	748	11	2611
TRGC	100	18752	2	2348	1106	436	62	95	1747	8	4103
CGC	100	18756	2	2309	1057	414	63	0	1548	11	3868
CDSR	100	25272	0	2490	1012	0	0	0	1013	14	3517

Paraffins				Instruction Cycles (×1000)							
Mode	Input N	Heap (Wds)	GCs	Id	Id RTS					Idle	Total
					Basic	Mark	Sweep	TREC	Total		
TRGC	10	8870	2	678	352	123	30	11	528	20	1225
CGC	10	8870	2	681	354	89	30	0	481	16	1177
CDSR	10	10690	0	700	308	0	0	0	311	40	1051
TRGC	11	15760	2	963	538	286	52	19	905	40	1908
CGC	11	15760	2	964	538	185	52	0	782	38	1784
CDSR	11	17572	0	1000	462	0	0	0	465	87	1553
TRGC	12	28144	3	1482	900	620	93	38	1660	45	3187
CGC	12	28148	3	1487	902	387	93	0	1389	43	2920
CDSR	12	30722	0	1523	749	0	0	0	752	107	2383
TRGC	13	46884	3	2521	1607	2765	234	145	4763	121	7405
CGC	13	46884	3	2528	1608	1726	234	0	3576	114	6218
CDSR	13	58682	0	2566	1299	0	0	0	1302	292	4160

Figure 8.7: Performance Results for Quicksort and Paraffins.

For each of the programs we tested, we ran three versions: TRGC, CGC, and CDSR. The TRGC version is the program running with type-reconstructing garbage collection. The CGC version is running with conservative garbage collection, and the CDSR is the automatically annotated version running with no garbage collection. Both garbage collectors used the mark and sweep algorithm, and used the same implementation of sweeping and inter-processor synchronization. Using a simple GC algorithm allowed us to separate the basic heap management cost (allocation and deallocation) from the overall cost of garbage collection. Thus, the cost of object traversal and marking of TRGC and CGC can be truly ascribed to their respective object identification strategies.

In all three cases, actual heap storage management and statistics collection was performed by the same Id run-time system. Although statistics gathering was mildly intrusive, it constituted a tiny fraction of total cycles executed. Online statistics processing (re-sampling profiles) was not counted.

Gamteb				Instruction Cycles (×1000)								
Mode	Input N	Heap (Wds)	GCs	Id	Id RTS Basic	Mark	Sweep	TREC	Total	Idle	Total
TRGC	25	5634	3	1948	289	59	15	6	381	65	2394
CGC	25	5634	3	1950	291	102	15	0	417	58	2425
CDSR	25	1780	0	2000	313	0	0	0	316	56	2371
TRGC	50	11278	3	3837	586	73	30	8	709	123	4668
CGC	50	11278	3	3824	584	117	30	0	739	119	4682
CDSR	50	1780	0	3929	627	0	0	0	631	111	4671
TRGC	75	16812	2	5485	836	35	34	4	919	191	6594
CGC	75	16812	2	5490	840	62	34	0	942	172	6604
CDSR	75	1712	0	5628	906	0	0	0	913	173	6714
TRGC	100	22506	2	7150	1096	70	46	7	1228	246	8624
CGC	100	22506	2	7159	1101	97	45	0	1250	227	8636
CDSR	100	1840	0	7355	1191	0	0	0	1198	222	8775

Wavefront				Instruction Cycles (×1000)							
Mode	Input N	Heap (Wds)	GCs	Id	Id RTS Basic	Mark	Sweep	TREC	Total	Idle	Total
TRGC	10	1726	3	495	22	21	1	2	56	50	601
CGC	10	1726	3	495	22	49	1	0	79	50	624
CDSR	10	1078	0	518	21	0	0	0	23	48	589
TRGC	20	5256	5	1772	41	41	2	3	100	224	2096
CGC	20	5316	5	1769	41	242	2	0	295	224	2288
CDSR	20	1300	0	1821	40	0	0	0	42	211	2074
TRGC	30	10500	5	3922	65	41	2	3	124	523	4569
CGC	30	10500	5	3921	65	462	2	0	540	524	4985
CDSR	30	6548	0	4064	64	0	0	0	66	494	4624
TRGC	40	20680	5	6946	113	41	2	3	172	945	8064
CGC	40	20740	5	6934	113	832	2	0	957	947	8839
CDSR	40	12692	0	7191	112	0	0	0	114	893	8198

Figure 8.8: Performance Results for Gamteb and Wavefront.

We simulated several problem sizes on a single processor with each program and storage management scheme. Figure 8.7 and Figure 8.8 show the performance results for each of the benchmarks. The first two columns identify the storage management scheme (Mode) and the input size (N). The next two columns show the maximum heap size used (Heap) during each run measured in 32-bit words, and the number of garbage collections performed (GCs). Subsequent columns record timing information for various categories of instructions measured in Kcycles. In each of the garbage collected runs, the run-time system initiated the garbage collection when the currently allocated space exceeded 75% of the total heap space. Garbage collection was switched off for CDSR runs.

The timing information for each benchmark run is broken up into several categories. The amount of time spent in Id computation threads (Id) includes basic computation work, math-library subroutine calls, split-phase memory referencing and program I/O. The time spent in the run-time system (Id RTS) is classified into the time spent in basic storage management

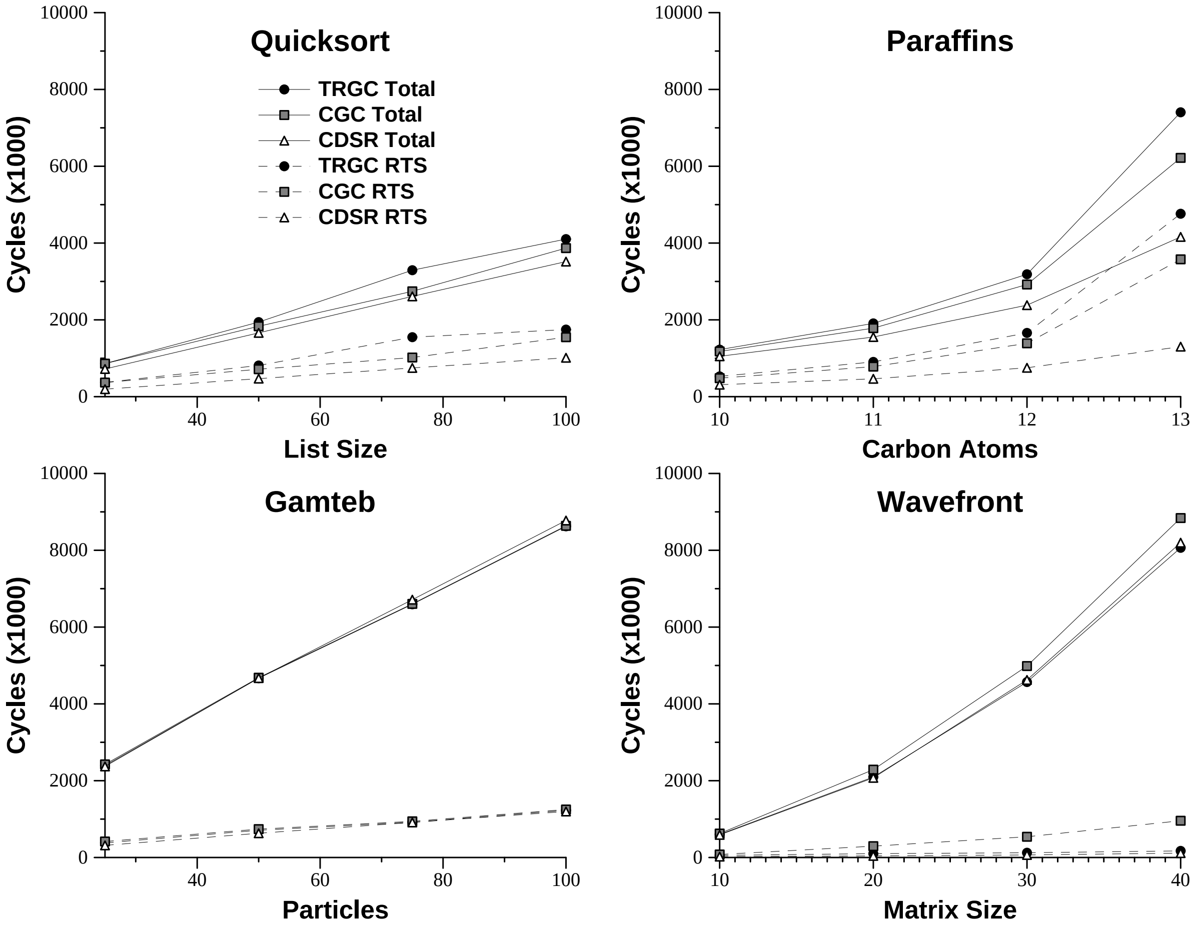

Figure 8.9: Total Cost and Run-time System Cost for the Benchmarks.

(allocation/deallocation), frame and object marking during garbage collections, object sweeping, and type reconstruction. The remaining time is spent idling through the scheduling loop waiting for messages to arrive through the network.[8]

8.6.2 Performance Analysis

Time Analysis

The total instruction cycles and the cycles spent in the run-time system (including garbage collection) for all the runs are summarized in Figure 8.9. These curves give an idea of the growth of run-time system cost of the various schemes as a function of problem size and as a fraction of the total cost.

Several trends are apparent from Figure 8.9. The CDSR scheme consistently has the lowest run-time cost since it does not perform any garbage collection and only incurs the basic heap and frame management cost (allocation and deallocation). The fraction of time spent in the

[8]Even if only a single processor is used out of a multi-processor *T configuration, all messages are sent out to the network and received after some delay. This may cause idle cycles on the processor if it does not have anything else to do.

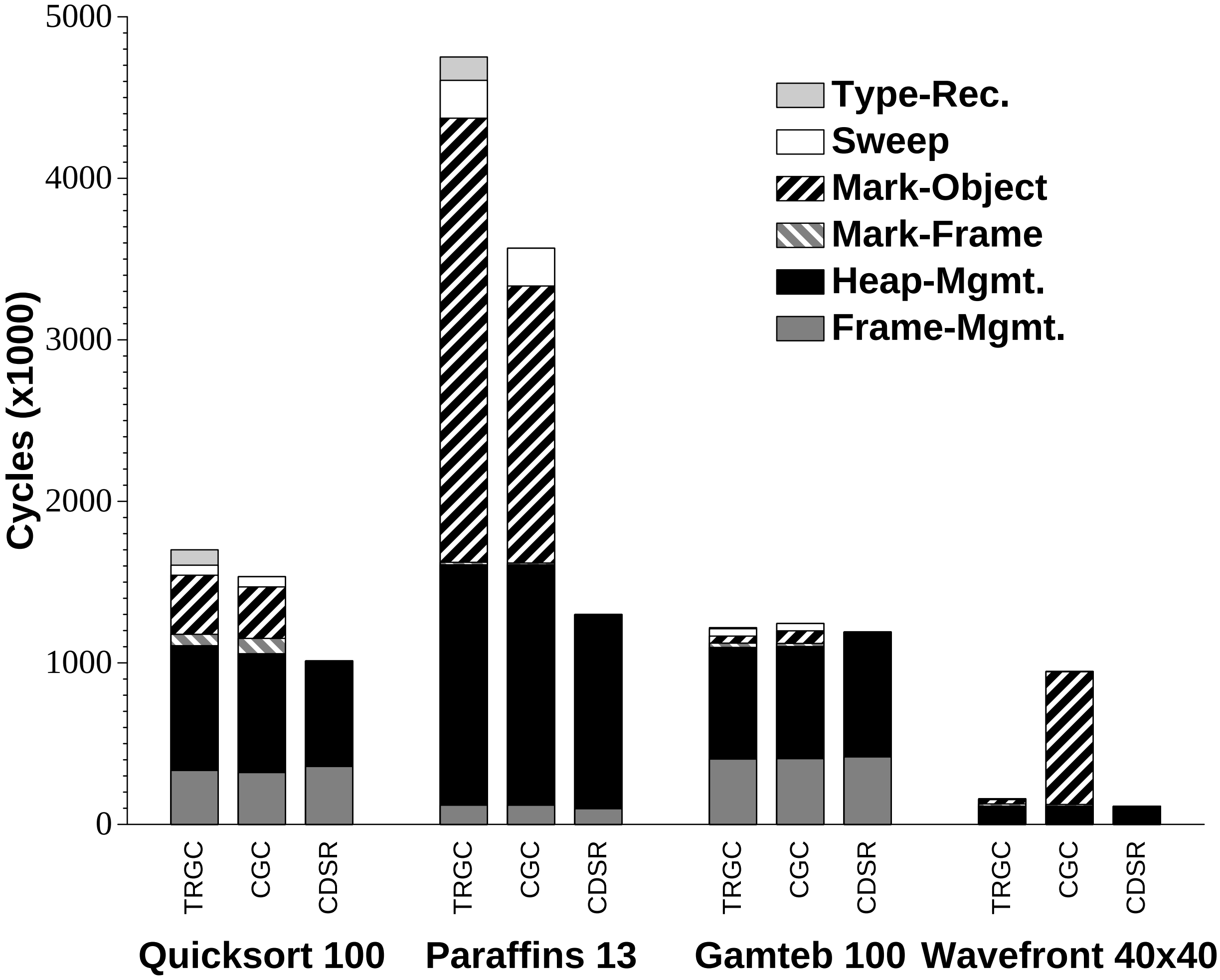

Figure 8.10: Run-time System Cost Breakup.

run-time system varies widely depending upon the nature of the application and the cost and the number of garbage collections performed. For example, Paraffins allocates a lot of small-sized data-structures keeping them live until the very end. Thus, each mark phase has to do a lot of work. Similarly, Quicksort rapidly unfolds into a tree of activation frames each of which holds onto a substantial amount of storage, so the cost of marking is high there as well. On the other hand, for Gamteb, the size of the live heap is quite small so the garbage collected schemes incur very little overhead.

Comparing the relative run-time costs of TRGC and the CGC, we find that for Quicksort and Paraffins, TRGC does worse than CGC, while for Wavefront TRGC performs better. This wide variation can be explained by examining the run-time cost breakup shown in Figure 8.10 for the largest sized runs. We split the basic storage management cost shown in Figure 8.7 and Figure 8.8 between the cost of managing the frame area and the cost of managing the heap. The marking cost is similarly split between the cost of marking the frames and the cost of marking the live heap objects.

Looking at Figure 8.10, TRGC spends a significant amount of time in the type reconstruction phase for both Quicksort and Paraffins. This is because both these benchmarks contain several polymorphic functions. Thus, the type reconstruction mechanism has to generate and propagate the exact run-time type instantiation down from the root to each polymorphic frame in the dynamic call tree. On the other hand, the type reconstruction cost is hardly visible in Gamteb and Wavefront that are not polymorphic and largely consist of first-order functions. Furthermore, during type reconstruction and interpreted marking, the run-time types are represented as C data-structures and are currently managed using conventional `malloc` and `free` system calls. This cost can be substantially reduced by using a specialized version of `malloc`.

The marking cost of TRGC is also about 1.5-2.2 times higher than that of CGC in case of Quicksort of 100 elements and Paraffins of 13 carbon atoms. Our current implementation interprets the type structures at run-time in order to traverse and mark the corresponding run-time objects. This interpretation overhead could be eliminated by using the compiled marking schema as described in Section 8.4.2 where the compiler generates a specialized marking routine for each source type parameterized over its polymorphic variables. Furthermore, these routines can be inlined to produce highly optimized traversal and marking functions for each user-defined function activation frame.

In the case of Wavefront, TRGC takes much less time than CGC, and very little more time in total than CDSR, where no marking at all took place. For Wavefront of 40×40, the marking cost of CGC is 25 times higher than that of TRGC. TRGC did so well because it could determine that the arrays contained only scalar data by inspecting their run-time type. Therefore, it only marked the arrays themselves and did not scan for pointers inside them, as CGC did. This scanning cost depends on the total size of the arrays and was responsible for the quadratic growth in run-time cost for CGC as shown in Figure 8.9. However, sweeping took the same amount of time for both TRGC and CGC.

The wavefront example shows that in an ideal situation, the time to mark the heap for TRGC is proportional to the total *number* of live object references, rather than the total *amount* of live storage as it is for CGC. TRGC can use the reconstructed type information to avoid scanning elements of scalar arrays and scalar fields within records and algebraic types.

Space Analysis

In terms of space usage, both TRGC and CGC perform identically. As shown in Figure 8.7 and Figure 8.8, both TRGC and CGC perform the same number of garbage collections in all runs and use roughly the same amount of heap storage. Both TRGC and CGC runs were provided with the same amount of initial storage. Although, the size of the initial storage was kept sufficiently large to avoid thrashing. This accounts for the small number of garbage collections performed.

Each garbage collected run also performed a final GC at the end of the run to reclaim all the uncollected garbage. Due to this final garbage collection, the TRGC and CGC runs actually reclaimed more storage than the CDSR runs, because the compiler could not insert deallocation commands for all of the temporary storage.

CGC is able to reclaim all the garbage because of our restrictive compilation model and support from the run-time system. As mentioned earlier, in our system all actual pointers directly point to the head of a heap object. This not only reduces the overhead of guessing whether a given value is a valid heap pointer or not but also avoids creating many more ambiguous pointers for the garbage collector to check for. The run-time system further eliminates the chances of making the wrong guess by marking the head of every object with a special bit-pattern.

The performance of CDSR varies with the application. For Gamteb and Wavefront, CDSR is able to insert deallocation commands to reclaim all the garbage automatically. Therefore, these benchmarks are able to run under CDSR without leaking any storage. The garbage collected versions for these benchmarks had to be given 2-10 times the storage used by CDSR to avoid thrashing. On the other hand, for Paraffins and Quicksort, CDSR is able to reclaim only 10-20% of the total garbage, therefore the TRGC and CGC versions are able to run in same or less storage than the CDSR version without thrashing. This shows that in general, CDSR may need additional storage reclamation support from an independent garbage collector, although it works very efficiently for applications where data-structures are easily analyzed.

8.7 Conclusions

In this chapter, we have described a direct application of complete run-time type reconstruction, namely, tagless garbage collection (TRGC). We used the reconstruction algorithm described in Chapter 7 to reconstruct the exact types of all run-time objects. We also described an interpreted and a compiled marking schema for traversing and marking live run-time objects using the reconstructed type information. We have implemented the interpreted marking schema on a simulator for the *T architecture and compared its performance with conservative garbage collection (CGC) and compiler-directed storage reclamation (CDSR) on several benchmarks.

Our results show that in general, TRGC does more work in marking the live objects than CGC, unless it can avoid scanning large, scalar, array-like objects using type information. The type reconstruction overhead increases with the amount of polymorphism and higher-order functions (closures) used in the program, although the cost of reconstruction is small compared to the cost of marking live objects with type interpretation. The cost of interpreted marking itself should get reduced considerably using the compiled marking schema instead of type interpretation.

TRGC has the additional advantage that other storage reclamation schemes may be used, such as compaction or copying. These may not be used with CGC because they require updating live pointers, and CGC cannot guarantee that what it uses as a pointer is not really a scalar value. On the other hand, TRGC requires initialization of polymorphic and pointer data with valid values and cannot cope with stale data as CGC can.

CDSR consistently does better than either of the garbage collection schemes in terms of time spent in the run-time system. This is as expected, although sometimes it is not able to collect all the garbage and therefore requires more memory than strictly necessary. CDSR also takes much longer to compile, sometimes increasing compile-time by a factor of 10.

On the whole, type reconstruction and type-reconstruction-based garbage collection seem to be a promising area of research with a lot of scope for compiler optimization and run-time performance improvement. This initial study has shown that type reconstruction based garbage collection is certainly feasible and can be competitive with other storage management strategies under the right mix of applications.

8.7.1 Future Work

There are several dimensions in which further investigation would be useful. The first step would be to implement the compiled marking schema and compare its performance with our current interpreted marking schema. We expect to see a substantial improvement in performance using specialized marking functions. Our experience also shows that mixed storage management schemes that combine garbage collection with explicit storage reclamation within the same run-time environment are feasible and may be able to combine the benefits of both schemes running on its own.

Although our system has been designed and implemented for a multi-processor architecture, we have currently made a study for only a single processor. We would like to see how TRGC scales under a multi-processor environment and quantify the inter-processor communication overhead for type reconstruction.

It would be very interesting to compare the performance of TRGC with an explicitly tagged object identification scheme implemented within the same framework. It would be interesting to know if TRGC offers any concrete advantages over that technique.

Finally, it would be useful to implement a compacting garbage collector based on type reconstruction with a very simple allocation scheme (bumping a pointer) and compare its heap management overhead with that of the CGC and CDSR that require a more sophisticated storage management scheme (free-lists).

Bibliography

[AA91] Zena M. Ariola and Arvind. Compilation of Id. In *Proceedings of the fourth Workshop on Languages and Compilers for Parallel Computing, Santa Clara, California*, August 1991. Also available as CSG Memo 341, MIT Lab. for Computer Sc., Cambridge, MA 02139.

[AA93] Zena M. Ariola and Arvind. Graph Rewriting Systems: Capturing Sharing of Computation in Language Implementations. Computation Structures Group Memo 347, MIT Laboratory for Computer Science, 545 Technology Square, Cambridge, MA 02139, April 1993.

[AA94] Zena M. Ariola and Arvind. Properties of a First-order Functional Language with Sharing. CSG Memo 347-1, Laboratory for Computer Science, MIT, Cambridge, MA 02139, June 1994. To appear in Theoretical Computer Science, September 1995.

[AC93] Shail Aditya and Alejandro Caro. Compiler-directed Type Reconstruction for Polymorphic Languages. In *Proceedings of the ACM Conference on Functional Programming Languages and Computer Architecture, Copenhagen, Denmark*, pages 74–82, June 1993.

[AFH94] Shail Aditya, Christine H. Flood, and James E. Hicks. Garbage Collection for Strongly-Typed Languages using Run-time Type Reconstruction. In *Proceedings of the ACM Conference on Lisp and Functional Programming, Orlando, Florida, USA*, pages 12–23. ACM Press, June 1994.

[AHU74] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *The Design and Analysis of Computer Algorithms.* Addison-Wesley, 1974.

[AM89] Andrew W. Appel and David B. MacQueen. *Standard ML Reference Manual.* Princeton University and AT&T Bell Laboratories, Preliminary edition, 1989. Distributed along with the Standard ML of New Jersey Compiler.

[ANP89] Arvind, Rishiyur S. Nikhil, and Keshav K. Pingali. I-Structures: Data Structures for Parallel Computing. *ACM Transactions on Programming Languages and Systems*, 11(4):598–632, 1989.

[App89] Andrew W. Appel. Runtime tags aren't necessary. *Lisp and Symbolic Computation*, 2(2):153–163, June 1989.

[App90] Andrew W. Appel. A runtime system. *Lisp and Symbolic Computation*, 3(4):343–380, November 1990.

[Bar88] Joel F. Bartlett. Compacting Garbage Collection with Ambiguous Roots. Research Report 88/2, Western Research Laboratory, Digital Equipment Corporation, February 1988.

[Bar92] Paul S. Barth. *Atomic Data Structures for Parallel Computing.* PhD thesis, Laboratory for Computer Science, MIT, Cambridge, MA 02139, March 1992. Available as Technical Report MIT/LCS/TR-532.

[Bec92] Michael J. Beckerle. An Overview of the START(*T) Computer System. Motorola Technical Report MCRC-TR-28, Motorola Cambridge Research Center, One Kendall Square, Building 200, Cambridge, MA 02139, July 1992.

[Blo89] A. Bloss. Update analysis and the efficient implementation of functional aggregates. In *Proceedings of the ACM Conference on Functional Programming Languages and Computer Architecture, London, UK.* ACM, September 1989.

[BNA91] Paul S. Barth, Rishiyur S. Nikhil, and Arvind. M-Structures: Extending a Parallel, Non-Strict, Functional Language with State. In *Proceedings of the ACM Comference on Functional Programming Languages and Computer Architecture*, pages 538–568. Springer-Verlag, 1991. LNCS 523.

[Bur77] Rod M. Burstall. Design Considerations for a Functional Programming Language. In *Infotech State of the Art Conference: The Software Revolution*, October 1977.

[BW88] H.-J. Boehm and M. Weiser. Garbage Collection in an Uncooperative Environment. *Software—Practice and Experience*, 18(9):807–820, September 1988.

[Car89] L. Cardelli. Typeful Programming. In E. J. Neuhold and M. Paul, editors, *Formal Description of Programming Concepts*, pages 431–507. Springer-Verlag, 1989.

[Car93] Alejandro Caro. A Debugger for Id. Master's thesis, Massachusetts Institute of Technology, February 1993.

[CCF+93] Derek Chiou, Alejandro Caro, Christine Flood, James E. Hicks, and Michael J. Beckerle. Run time support for Id running on *T, version 1.4. Computation structures group memo, MIT, Laboratory for Computer Science, 545 Technology Square, Cambridge, MA 02139, May 1993.

[Dam85] L. Damas. *Type Assignment in Programming Languages.* PhD thesis, University of Edinburgh, Department of Computer Science, 1985.

[Dar77] John Darlington. Program Transformation and Synthesis: Present Capabilities. Research Report 77/43, Department of Computing and Control, Imperial College of Science and Technology, London, September 1977. Also appears as Report No. 48, Department of Artificial Intelligence, University of Edinburgh.

[DM82] L. Damas and R. Milner. Principal Type-schemes for Functional Programs. In *Proceedings of the 9th Symposium on Principles of Programming Languages*, pages 207–212, January 1982.

[FLR+94] Babak Falsafi, Alvin R. Lebeck, Steven K. Reinhardt, Ioannis Schoinas, Mark D. Hill, James R. Larus, Anne Rogers, and David A. Wood. Application-Specific Protocols for User-Level Shared Memory. In *Supercomputing '94, Proceedings.* IEEE Computer Society Press, November 1994.

[GG92] Benjamin Goldberg and Michael Gloger. Polymorphic Type Reconstruction for Garbage Collection without Tags. In *Proceedings of the ACM Conference on Lisp and Functional Programming*, pages 53–65, 1992.

[GJLS87] David K. Gifford, Pierre Jouvelot, John M. Lucassen, and Mark A. Sheldon. FX-87 Reference Manual. Technical Report MIT/LCS/TR-407, MIT Laboratory for Computer Science, September 1987.

[GJSO91] David K. Gifford, Pierre Jouvelot, Mark A. Sheldon, and James W. O'Toole. Report on the FX-91 Programming Language. Technical Report MIT/LCS/TR-531, MIT Laboratory for Computer Science, 1991.

[Gol91] Benjamin Goldberg. Tag-free garbage collection for strongly typed programming languages. In *SIGPLAN '91 Conference on Programming Language Design and Implementation*, pages 165–176, June 1991.

[GP90] Benjamin Goldberg and Young Gil Park. Higher Order Escape Analysis: Optimizing Stack Allocation in Functional Program Implementations. In *Proceedings of the 3rd European Symposium on Programming*, pages 152–160. Springer-Verlag, 1990. LNCS 432.

[GPG91] Young Gil Park and Benjamin Goldberg. Reference Escape Analysis: Optimizing Reference Counting based on the Lifetime of References. In *Proceedings of the ACM Symposium on Partial Evaluation and Semantics-based Program Manipulation, Yale University, New Haven, CT, USA*, pages 178–189. ACM Press, June 1991.

[Gup90] Shail Aditya Gupta. An Incremental Type Inference System for the Programming Language Id. Master's thesis, Laboratory for Computer Science, MIT, Cambridge, MA 02139, September 1990. Available as Technical Report MIT/LCS/TR-488.

[HI89] W.L. Harrison III. The interprocedural analysis and automatic parallelization of scheme programs. *Lisp and Symbolic Computation*, 2(3-4):179–396, 1989.

[Hic93] James E. Hicks. Experiences with compiler-directed storage reclamation. In *Conference on Functional Programming Languages and Computer Architecture*, 1993.

[HJ92] James E. Hicks Jr. *Compiler-directed Storage Reclamation using Object Lifetime Analysis.* PhD thesis, Laboratory for Computer Science, MIT, Cambridge, MA 02139, 1992. Available as Technical Report MIT/LCS/TR-555.

[HM93] R. Harper and J. C. Mitchell. On the Type Structure of Standard ML. *ACM Transactions on Programming Languages and Systems*, 15:211–252, April 1993.

[HMV93] My Hoang, John Mitchell, and Ramesh Viswanathan. Standard ML weak polymorphism and imperative constructs. In *Proceedings of the Eighth Annual Symposium on Logic in Computer Science*, pages 15–25. ACM Press, June 1993.

[Hud92] Paul Hudak. Mutable Abstract Datatypes -or- How to Have Your State and Munge It Too. Research Report YALEU/DCS/RR-914, Department of Computer Science, Yale University, New Haven, CT 06520, December 1992. Revised May 1993.

[HWe90] P. Hudak and P. Wadler (editors). Report on the programming language Haskell, a non-strict purely functional language (Version 1.0). Technical Report YALEU/DCS/RR777, Yale University, Department of Computer Science, April 1990.

[JG91] Pierre Jouvelot and David K. Gifford. Algebraic Reconstruction of Types and Effects. In *Proceedings of the 1991 ACM Conference on Principles of Programming Languages*, pages 303–310. ACM, 1991.

[Joh85] Thomas Johnsson. Lambda lifting: Transforming programs to recursive equations. In *Springer-Verlag LNCS 201 (Proc. Functional Programming Languages and Computer Architecture, Nancy, France)*, September 1985.

[JW75] Kathleen Jensen and Niklaus Wirth. *PASCAL User Manual and Report.* Springer-Verlag, 1975.

[KR88] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language.* Prentice Hall, second edition, 1988.

[LAB+81] Barbara Liskov, Russell Atkinson, Toby Bloom, Eliot Moss, J. Craig Schaffert, Robert Scheifler, and Alan Snyder. *CLU Reference Manual*, volume 114 of *Lecture Note in Computer Science.* Springer-Verlag, 1981.

[Ler92] Xavier Leroy. Polymorphic Typing of an Algorithmic Language. Rapports de Recherche 1778, INRIA, Rocquencourt, France, October 1992. English translation of the author's Ph.D. thesis originally in French.

[Ler93] Xavier Leroy. Polymorphism by name for references and continuations. In *Proceedings of the ACM Symposium on Principles of Programming Languages.* ACM Press, 1993.

[LG88] John M. Lucassen and David K. Gifford. Polymorphic Effect Systems. In *Proceedings of the Fifteenth Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming languages, San Diego, California*, pages 47–57, January 1988.

[LPJ94] John Launchbury and Simon L. Peyton Jones. Lazy Functional State Threads. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation, Orlando, Florida, USA.* ACM, June 1994.

[Luc87] John M. Lucassen. *Types and Effects – Towards the Integration of Functional and Imperative Programming.* PhD thesis, Laboratory for Computer Science, MIT, Cambridge, MA 02139, August 1987. Available as Technical Report MIT/LCS/TR-408.

[LW91] Xavier Leroy and Pierre Weis. Polymorphic type inference and assignment. In *Proceedings of the ACM Symposium on Principles of Programming Languages*, pages 291–302. ACM, January 1991.

[Mai90] Harry G. Mairson. Deciding ML Typability is Complete for Deterministic Exponential Time. In *Proceedings of the 17th ACM Symposium on Principles of Programming Languages*, pages 382–401, January 1990.

[Mil78] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, 1978.

[MT91] Robin Milner and Mads Tofte. *Commentary on Standard ML.* The MIT Press, Cambridge, Massachusetts, 1991.

[MTH90] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML.* The MIT Press, Cambridge, Massachusetts, 1990.

[NAH93] Rishiyur S. Nikhil, Arvind, and James Hicks. pH Language Proposal (Preliminary). Circulated on the pH mailing list, September 1993.

[Nik91] Rishiyur S. Nikhil. Id Language Reference Manual Version 90.1. Technical Report CSG Memo 284-2, MIT Laboratory for Computer Science, 545 Technology Square, Cambridge, MA 02139, July 15 1991.

[Nik94] Rishiyur S. Nikhil. *Cid*: A Parallel, "Shared-memory" C for Distributed-memory Machines. In *Proceedings of the 7th Annual Workshop on Languages and Compilers for Parallel Computing, Ithaca, NY.* Cornell Theory Center, Cornell University, August 1994.

[NPA92] Rishiyur S. Nikhil, Gregory M. Papadopoulos, and Arvind. *T: A Multithreaded Massively Parallel Architecture. In *Proceedings of the 19th International Symposium on Computer Architecture, Queensland, Australia.* ACM Press, May 1992.

[OJ91] James William O'Toole Jr. Type Abstraction Rules for References: A comparison of four which have achieved notoriety. Technical Memo MIT/LCS/TM-390, Laboratory for Computer Science, MIT, 545 Technology Square, Cambridge, Massachusetts 02139, August 1991.

[PBGB93] G. M. Papadopoulos, G. A. Boughton, R. Greiner, and M. J. Beckerle. *T: Integrated building blocks for parallel computing. In *Proceedings of Supercomputing '93*, 1993.

[PJ92] Simon L. Peyton Jones. Implementing lazy functional languages on stock hardware: the Spineless Tagless G-machine. *Journal of Functional Programming*, 2(2):127–202, April 1992.

[PJW92] Simon L. Peyton Jones and Philip Wadler. A static semantics for Haskell, February 1992.

[PJW93] Simon L. Peyton Jones and Philip Wadler. Imperative Functional Programming. In *Proceedings of the 20th ACM Symposium on Principles of Programming Languages, Charleston, South Carolina, USA*, pages 71–84. ACM, January 1993.

[Pla92] P.J. Plauger. *The Standard C Library.* Prentice Hall, Englewood Cliffs, New Jersey 07632, 1992.

[Plo81] G. D. Plotkin. A Structural Approach to Operational Semantics. Technical Report DAIMI FN-19, Computer Science Department, Aarhus University, Aarhus, Denmark, September 1981.

[Rey74] J. C. Reynolds. Towards a Theory of Type Structure. In *Paris Colloquium on Programming*, volume 19 of *Lecture Notes in Computer Science*, pages 408–425. Springer-Verlag, 1974.

[Rob65] J. A. Robinson. A Machine-Oriented Logic Based on the Resolution Principle. *Journal of the ACM*, 12(1):23–41, 1965.

[SJ90] Guy L. Steele Jr. *Common Lisp: The Language*. Digital Press, second edition, 1990.

[TJ92] Jean-Pierre Talpin and Pierre Jouvelot. The Type and Effect Discipline. In *Proceedings of the ACM Symposium on Logic in Computer Science*, pages 162–173. ACM Press, 1992.

[Tof90] Mads Tofte. Type Inference for Polymorphic References. *Information and Computation*, 89:1–34, 1990.

[Tol94] Andrew Tolmach. Tag-free Garbage Collection Using Explicit Type Parameters. In *Proceedings of the 1994 ACM Conference on Lisp and Functional Programming*, pages 1–11. ACM Press, June 1994.

[Tra86] Kenneth R. Traub. A Compiler for the MIT Tagged-Token Dataflow Architecture. Master's thesis, Laboratory for Computer Science, MIT, Cambridge, MA 02139, August 1986. Available as Technical Report MIT/LCS/TR-370.

[TT93] Mads Tofte and Jean-Pierre Talpin. A Theory of Stack Allocation in Polymorphically Typed Languages. Technical Report 93/15, Department of Computer Science (DIKU), Copenhagen University, 1993.

[Tur85] D. A. Turner. Miranda: A non-strict functional language with polymorphic types. In *Lecture notes in Computer Science*, volume 201. Springer Verlag, September 1985.

[Wad90] Philip Wadler. Linear types can change the world! In *Proceedings of the Working Conference on Programming Concepts and Methods, Israel*, pages 385–407. North-Holland, 1990.

[WB89] Philip Wadler and Stephen Blott. How to make *ad-hoc* polymorphism less *ad hoc*. In *Proceedings of the 16th ACM Symposium on Principles of Programming Languages, Austin, Texas*, pages 60–76, January 1989.

[Wil92] Paul R. Wilson. Uniprocessor Garbage Collection Techniques. In *Proceedings of the International Workshop on Memory Management, St. Malo, France*, pages 1–42. Springer-Verlag, September 1992. LNCS 637.

[Wri92] Andrew K. Wright. Typing References by Effect Inference. In *Proceedings of the 4th European Symposium on Programming, Rennes, France*, pages 473–491. Springer-Verlag, February 1992. Lecture Notes in Computer Science, volume 582.

[Wri93] Andrew K. Wright. Polymorphism for Imperative Languages without Imperative Types. Technical Report TR93-200, Rice University, February 1993.