Destructive Effect Analysis And Finite Differencing For Strict Functional Languages

by

Chung Yung

A dissertation submitted in partial fulfillment

of the requirements for the degree of

Doctor of Philosophy

Department of Computer Science

New York University

September 1999

Approved: _____

Benjamin Goldberg

To the memory of my grandfather, HaiLing Yung (1918-1995)

Acknowledgements

I am most grateful to my advisor, Professor Benjamin Goldberg, not only for his guiding me through the doctoral study but also for his continued support in various aspects. His deep involvement have been invaluable during the preparation of this thesis.

I would like to thank Professor Edmond Schonberg and Professor Anindya Banerjee for being on my thesis committee. Their comments are precious and very helpful for this thesis.

I appreciate Professor Robert Paige for showing me some important work in this area.

I would like to thank Professor Malcolm Harrison of NYU Griffin Project, and Professor Ajit Kambil of NYU EDGAR Project. Their leadership of the research projects made it very pleasant to work with them. Much of the motivation of this dissertation came from the experience of working in both projects.

I would like to thank my officemate Madhu Nayakkankuppam. The talks and

discussions in our office have been an important part of my doctoral study. Especially, I appreciate his kindness of letting me use his C code for measuring benchmarks.

I am grateful to Allan Leung for his comments to the design of the experimental language. I would like to thank Nick Afshartous and Dao-I Lin for sharing their experiences.

I would like to thank my parents, Chiahsien Yung and Yuchu K. Yung, and my grandmother Rueyti W. Yung for their unconditional support over the years.

Preface

Destructive update optimization is critical to the performance of functional programs. Pure functional languages do not allow mutations, destructive updates, or selective updates so that the straightforward implementations of functional languages induce large amounts of copying to preserve the program semantics. The unnecessary copying of data can increase both the execution time and the memory requirements of an application. Destructive update optimization makes an essential improvement to the implementation of functional programs with compound data structures, such as arrays, sets, and aggregates. Moreover, for many of the compiler optimization techniques that depend on the side-effects, destructive update analysis provide the input for applying such optimization techniques to functional programs.

Among other compiler optimization techniques, finite differencing captures common yet distinctive program constructions of costly repeated calculations and transforms them into more efficient incremental program constructions. Finite dif-

ferencing generalizes the classical method of *reduction in operator strength*. When applied to set-theoretic expressions, this technique can transform algorithms from a high-level concise but inefficient version into a more complex but efficient version. Since finite differencing relies on side-effects to improve the performance of programs, when we apply finite differencing together with the destruction update optimization to functional programs we may substantially improve the performance.

In this dissertation, we develop a new approach to destructive update analysis, called *destructive effect analysis*. We present the semantic model and the abstract interpretation of destructive effect analysis. We design *EAS*, an experimental applicative language with set expressions. The implementation of the destructive effect analysis is integrated with the optimization phase of our experimental compiler of EAS. We apply finite differencing to optimize pure functional programs, and we show the performance improvement that results from applying the finite differencing transformations together with the destructive update optimization.

This dissertation is organized in five chapters. We give a motivating example and the background of the dissertation in Chapter 1. Chapter 2 presents the semantic model of dynamic destructive effect analysis and the abstract interpretation framework. Chapter 3 describes the implementation work of the dissertation, which includes: the extension of typed λ-calculus to sets, the design of EAS, the implementation of the EAS dynamic optimizer and the EAS static analyzer. In

Chapter 4, we present the application of destructive effect analysis to the finite differencing optimization for functional programs. We present the concluding comments and the future directions in the last chapter.

Contents

List of Figures

Chapter 1

Introduction

Destructive update optimization is critical for writing scientific codes in functional languages [WC98]. Pure functional languages do not allow mutations, destructive updates, or selective updates so that the straightforward implementations of functional languages induce large amounts of copying to preserve the program semantics. The unnecessary copying of data can increase both the execution time and the memory requirements of an application. Destructive update optimization makes an essential improvement to the implementation of functional programs with compound data structures, such as arrays, sets, and aggregates.

Moreover, for many of the compiler optimization techniques that depend on the side-effects, destructive update analysis provide the input for applying such optimization techniques. Among other compiler optimization techniques, finite

differencing captures common yet distinctive program constructions of costly repeated calculations and transforms them into more efficient incremental program constructions [Pai81, PK82]. Since finite differencing relies on side-effects to improve the performance of programs, when we apply finite differencing together with the destruction update optimization to functional programs we may substantially improve the performance.

Functional languages, which originated with Lisp, owe much of their spirit to λ-calculus. Lisp was the first significant programming language to allow the computation of all partial recursive functions by purely applicative, or functional, programs. One feature of functional languages that goes beyond the applicative realm is the inclusion of primitives for what is variously referred to as "mutations," "destructive updates," or "selective updates." We shall refer to a functional language with or without mutation primitives as *impure* or *pure*, respectively [Pip97].

Without mutation primitives, binding compound values to objects in pure functional languages raises an important issue of efficiency. A naive implementation that strictly adheres to the applicative model must copy whenever data values are modified. However, the cost associated with copying large data aggregates such as arrays and sets can become prohibitive [FO95]. This leads to our investigation in analysis techniques for optimizing pure functional programs with aggregate updates.

According the definition given by Draghicescu and Purushothaman [DP93], we may describe the destructive update problem as follows.

> Given the expression $update(e_1,\ e_2,\ e_3)$, determine at compile time, if possible, that the object denoted by e_1 will not be referred after the $update$ is performed; in such a case a compiler can generate code to update the object destructively.

The goal of this dissertation is to investigate the extent to which compiler optimization techniques add something essential to functional languages with sets and other aggregates, and the extent to which they can make compilers for pure functional languages generate object code that manipulates sets as efficiently as the compilers for impure functional languages do.

1.1 A Motivating Example

We shall start with an example. Consider the functional program in Figure 1.1 that does topological sort. For a graph with a set of nodes $\{1, 2, 3\}$ and a set of edges $\{(1, 2), (1, 3), (2.3)\}$, the program will have a set $S = \{1, 2, 3\}$, and a set $sp = \{(1, \{2, 3\}), (2, \{3\})\}$. Note that the algorithm implemented by the program is not efficient, and we will demonstrate how finite differencing may improve its efficiency. Also, note that the loops in the algorithm are implemented as recursive functions in the ML-like program.

```
/* fnext computes {a in S1 | sp{a} * S1 = {} } */
 1. fun fnext {} S1 = {};
 2.   | fnext (S2 with a) S1 =
 3.         if sp{a} * S1 == {}
 4.             then (fnext S2 S1) with a;
 5.             else fnext S2 S1;

/* TSort: the loop in TopSort */
 6. fun TSort {} S L = L;
 7.   | TSort (T with a) S L =
 8.         let S1 = S less a;
 9.             T1 = fnext S1 S1;
10.          in TSort T1 S1 (a :: L);

/* TopSort is the main function for topological sort */
11. fun TopSort S =
12.     let T = fnext S S;
13.      in TSort T S [];
```

Figure 1.1: An ML-like functional program of topological sort

We assume that all the updating expressions are implemented in a naive and straightforward way. Since in each iteration it takes $O(nm)$ time to construct the set $T1$ in line 9, the complexity of this version of topological sort is $O(n^2m)$, where n is the number of nodes and m is the number of edges.

To improve the efficiency of the program in Figure 1.1, we may use Paige's finite differencing technique to optimize the program. When optimizing the program using finite differencing, we replace the code for constructing $T1$ in each iteration by the code for maintaining Zr from the value of Zr in previous iteration. The finite differencing transformations use two auxiliary sets, *prec* and *count*, for this purpose. The program that results from the finite differencing transformations is show in Figure 1.2.

Intuitively, since the program in Figure 1.2 replaces the expressions that construct new sets by the expressions that insert an element to a set, we expect the program in Figure 1.2 will be implemented with a better efficiency than the program in Figure 1.1. However, in pure functional languages, once an object is bound to a variable, the object cannot be changed. It is not clear how the compilers of pure functional language could generate efficient code for line 16

```
let count with <u, c> = C;
```

and for line 18

```
chkZ S <(Zr with u),(count with <u, 0>)>;
```

```
/* initTS: initialize prec and count set */
 1. fun initTS {} <S,C> = <S,C>;
 2.   | initTS (sp with <x, y>) <S,C> =
 3.       if y in DOM S
 4.           then let prec with <y, z> = S;
 5.                    count with <y, c> = C;
 6.                in initTS sp
    (<prec with <y, z with x>, count with <y,c+1>>);
 7.           else initTS sp (<S with <y,{x}>, C with <y,1>>);

/* initZ:  initialize Zrset */
 8. fun initZ {} C Zr = Zr;
 9.   | initZ (S with a) C Zr =
10.       let c = map C a;
11.        in if c == 0
12.           then initZ S C (Zr with a);
13.           else initZ S C Zr;

/* chkZ: the loop in TSort */
14. fun chkZ {} <Zr, C> = <Zr, C>;
15.   | chkZ (u with S) <Zr, C> =
16.       let count with <u, c> = C;
17.        in if c == 1
18.           then chkZ S <Zr with u, count with <u, 0>>;
19.           else chkZ S <Zr, count with <u, c-1>>;

/* TSort: the loop in TopSort */
20. fun TSort {} S C L = L;
21.   | TSort (Zr with a) S C L =
22.       let <Z1, C1> = chkZ S{a} <Zr, C>;
23.        in TSort Z1 S C1 (a :: L);

/* TopSort */
24. fun TopSort S =
25.     let <prec, count> = initTS sp <{},{}>;
26.         Zrset = initZ S count {};
27.      in TSort Zrset prec count [];
```

Figure 1.2: The result of finite differencing on the program in Figure 1.1.

if the compilers could not destructively update the sets C, Zr, and *count*. Hence, it is not clear whether the implementation of the finite differencing optimized program in Figure 1.2 will take less time than that of the original program in Figure 1.1.

This motivates our investigation in how we may improve the efficiency of pure functional programs with finite differencing. Our approach is that we introduce destructive update analysis to the compilers of pure functional languages so that the implementation of functional programs actually destructively updates the compound values while preserving the pure functional semantics. We will describe more of this in Section 1.6.

1.2 Related Work

The SETL programming language, developed at New York University [SDD86, Sny90], is the first programming language with a general purpose set data structure. The optimization of programs with set operations is discussed in Schwartz [Scw75a, Scw75b], and in Freudenberger et al. [FSS83]. Manens [Ber82] and S3L [Lac92] implemented λ-calculi with sets. Goubault [Gou94] described HimML, an extension of Standard ML [MTH90] with efficient polymorphic set-theoretic data structures.

The analysis for destructive update optimization has been studied in the lit-

erature before. One of the early works is that of Mycroft [Myc81]. In Hudak and Bloss [HB85], the problem is discussed in an operational model based on the graph reduction. An applicative-order language is treated in Hudak [Hud87] using an abstraction of reference counting. A related analysis is presented by Schmidt [Scm85, Scm88], also in an applicative-order setting. The problem is also discussed in Bloss [Blo89a, Blo89b] as an application of path analysis. A variation of path analysis is also used in Gopinath and Hennessy [GH89] for a language with call-by-value semantics. Draghicescu and Purushothaman [DP93] offer a solution to object sharing and evaluation order problems for lazy functional languages. Fitzgerald and Oldehoeft explore the destructive update analysis for true multi-dimensional arrays with a graph-based language as an intermediate form [FO95]. Odersky proposes effect analysis for a safe embedding of mutable data structure in functional languages [Ode91]. Wand and Clinger present the set constraints for an interprocedural destructive update analysis with time stamps [WC98].

Finite differencing was first developed by Paige [Pai81] as a high-level global program optimization method that captures a commonly occurring yet distinctive mechanism of program construction in which repeated costly calculations are replaced by inexpensive incremental counterparts. When finite differencing transformations are applied to the algorithms that are expressed as high-level, lucid but inefficient program versions, the transformed algorithms are materialized as more

complex but efficient program versions. This method generalizes John Cocke's method of *strength reduction*, and provides a convenient framework to implement a host of program transformations, including Earley's *iteration inversion* [PK82]. The finite differencing technique has been successfully applied to program transformation systems from imperative source languages to imperative target languages [CP93, Pai94]. The recent work catches the spirit of finite differencing and applies it to incremental computation systems [Liu96], and to semi-automatic program development systems [Smi90].

1.3 Dissertation Overview

This dissertation explores the use of destructive update analysis for set functional languages in order to apply finite differencing at the optimization phase of functional language compilers. This dissertation also explores the cooperation between finite differencing and the compilation of functional programs from the viewpoints of both the abstract level and the implementation level.

One of the benefits of functional programming is that its clean semantics makes programs amenable to very powerful transformations, resulting in significant optimizations. In this dissertation, we investigate finite differencing in its application to functional language compilation and functional program transformations. Our goal is to improve the efficiency of functional programs in the same way that finite

differencing improves the efficiency of imperative programs.

The traditional destructive update analyses on functional languages have concentrated on array constructs. Bloss extends the standard semantics using path information and analyzes for destructive updates based on the path information [Blo89a, Blo89b, Blo94]. Draghicescu and Purushothaman [DP93] offer a uniform solution to object sharing and evaluation order problems for lazy functional languages [DP93]. Odersky presents effect analysis for a safe embedding of mutable data structure in functional languages [Ode91]. They all apply, in different ways, abstract interpretation frameworks for a destructive update analysis. In this dissertation, we also apply an abstract interpretation framework for the destructive update analysis of strict functional languages with set expressions.

Turner, Wadler, and Mossin [TWM95] proposed a new idea of a destructive update analysis with their used-once analysis, which integrates the analysis with the type system. However, they did not include the details of how to make this idea practical. Wand and Clinger presented the idea of a destructive update analysis using time stamps [WC98]. But, they did not include the precise algorithms of the destructive update analysis. In this dissertation, we collect the information for our destructive analysis in the type system, when possible. In this way we keep the analysis phase of our approach simple. We analyze the expressions in a program with the time stamps attached to the expressions.

Finite differencing is a high-level compiler optimization technique. The finite differencing technique is, from a theoretical point of view, language independent and may be applied to imperative languages, functional languages, logic languages, and so on. The finite differencing technique has been successfully applied to imperative programming language transformations [CP93, Pai94]. However, the evidence is still missing that the finite differencing technique can be applied to functional languages. In this dissertation, we investigate the analysis needed for finite differencing to improve the performance, and apply the finite differencing transformations to functional programs for improving the performance.

In order to explore destructive update analysis for optimizing set functional programs and to apply finite differencing to functional languages, we divide the work of this dissertation into three major components:

1. Compiling functional programs with sets:

   A few techniques have been successfully applied to the analysis of functional programs for optimizations. In this dissertation, we are especially interested in their relation to the optimization of functional programs with sets.

2. Destructive update analysis for set functional languages:

   We are interested in a new approach that combines the following two techniques:

   - applying an abstract interpretation framework for a static destructive

update analysis, and

- collecting the information needed in the type system.

3. Applying finite differencing:

   We are interested in applying finite differencing together with the destructive update analysis to optimize functional programs with sets.

1.4 Set Functional Languages

The first step of this dissertation is designing a functional language with set expressions. We designed and implemented an experimental language, which is called EAS (meaning Experimental Applicative language with Sets). We describe the design of EAS in this section and more details can be found in [Yun97].

The Design of EAS

We designed EAS, an experimental functional language with set notations. We start by enriching the typed λ-calculus with set notations. Since all functional programming languages can be viewed as syntactic variations of λ-calculus, the set-enriched λ-calculus eventually shows that all functional languages can be enriched with set notations [Yun97].

In a sense of implementation, a set is an unordered list whose elements appear only once [FSS83, SDD86]. It allows us to extend the typed λ-calculus to sets in

list operation	set operation
$nil[t]$	$emp[t]$
e_1 :: e_2	e_1 with e_2
hd e	rep e
tl e	$e_1$less e_2
isnil e	isemp e

Figure 1.3: list operations and set operations

a way similar to how λ-calculus handles lists. Analogous to the list operations, we have a few set operations. Figure 1.3 shows our proposed set operations.

For the with and less operations, the implementation performs a membership test before adding and deleting an element, respectively. In other word, e_1 with e_2 does nothing if $e_2 \in e_1$. In a similar way, e_1 less e_2 does nothing if $e_2 \notin e_1$.

Note that the rep operation is different from the arb operation in SETL. The rep operation returns the representative of a set, while the arb operation in SETL returns an arbitrary member in a set. Hereby, the rep operation is a deterministic (and possibly implementation dependent) operation. In addition, we include the membership test operator, $\in$, whose syntax is e_1 in e_2.

The next step of this dissertation is to build the compiler for a functional language that implements the set-enriched typed λ-calculus. We will describe the implementation of the EAS compiler in Section 3.2. Some early implementation results have been presented in [Yun98].

1.5 Destructive Update Analysis

The goal of a destructive update analysis is to eliminate unnecessary copying in an implementation while preserving the semantics. The conventional study on destructive update analysis has concentrated on the manipulations of the data structures like arrays. Most of the approaches apply the abstract interpretation framework for a static analysis. We will apply the technique of abstract interpretation to the destructive update analysis for functional languages with sets.

The used once analysis by Turner, Wadler and Mossin extends the Hindley-Milner type system with *uses*, yielding a type-inference based program analysis which determines when values are accessed at most once [TWM95]. We will describe the idea of integrating part of destructive update analysis into the type system.

1.5.1 Abstract Interpretation Framework

A few methods use abstract interpretation frameworks for a static destructive update analysis. We start with a brief description of three well-known methods of destructive update analysis: Bloss' path analysis, Draghicescu and Purushothaman's order of evaluation analysis, and Odersky's effect analysis.

Bloss presents an exact non-standard semantics, called *path semantics*, that models the order of evaluation in a non-strict sequential functional language, and

its computable abstraction *path analysis* [Blo89a, Blo89b]. It was shown that the information inferred by path analysis can be used to implement the destructive aggregate update optimization, in which the updates on aggregates that are provably not live are implemented destructively [Blo94].

Draghicescu and Purushothaman give a destructive update algorithm by applying their *reduction to variables* analysis [DP93], which is a method of detecting the variables that denote locations where the result of expressions might be stored.

Odersky proposed *effect analysis* to show the safety of embedding mutable data structures in functional languages [Ode91]. He developed a static criterion-based abstract interpretation which checks that any side-effect which a function may exert via a destructive update remains invisible.

It is interesting to note that all the three approaches apply the abstract interpretation frameworks. By viewing set as a first-order construct, we may apply a similar approach for the destructive update analysis of set programs.

1.5.2 Enriched Type System

Turner, Wadler and Mossin presented a method for determining when a value is used at most once [TWM95]. Their method is based on a modification of the Hindley-Milner type system. Each type is labeled to indicate whether the corresponding value is used at most once or may possibly be used many times. They

conjecture that their used-once analysis can be applied to the destructive update analysis of data structures. However, they do not include, in [TWM95], the details how this can be done.

We are interested in extending their idea of enriching the type system for a destructive update analysis. More specifically, we want to integrate the phase of information collection in our analysis into the type system in a way that Turner, Wadler and Mossin did the used-once type system.

In order to keep our type system simple, we analyze the liveness of values in the optimization phase of a compiler instead of in the type system. Therefore, in contrast to the used-once type system which does all the analysis in the type system, our enriched type system collects information and the destructive update analysis is performed in an analysis phase of the compiler, working with the information collected from the enriched type system.

1.6 Finite Differencing

When compared with some other program transformation techniques, finite differencing shows its advantages in several ways [PK82]:

1. Finite differencing may be applied over a large spectrum of language levels from high-level specifications to low-level program representations;

2. Finite differencing can converge swiftly from a very high-level inefficient form

of an algorithm to a much lower-level version of a more efficient implementation;

3. Finite differencing can be implemented systematically; and,

4. Finite differencing can be shown to yield asymptotic speedup.

Finite differencing has been applied to a few different areas, including program transformations between imperative programming languages [CP93, Pai94], semantic transformations on Lisp programs [Liu96], and semi-automatic program development systems [Smi90].

In the example of topological sort shown in Figure 1.1, it is not clear if applying finite differencing to the functional program may produce asymptotic speedup or not since the purely functional semantics may induces copying for the update operations. Our approach is to apply the finite differencing optimization together with the destructive update optimization. With destructive update optimization, we guarantee that copying is introduced only if it is necessary so that the result program of finite differencing transformation, shown in Figure 1.2, will produce asymptotic speedup.

1.7 Summary

The overall project of this dissertation is divided into the following tasks:

1. Implementation of a compiler for functional programs with set operations,
2. Definition of the semantics for the set functional language,
3. Development of a dynamic destructive update optimizer for the set functional language,
4. Development of a static destructive update analyzer for the set functional language,
5. Development and implementation of the finite differencing algorithms for functional programs, and
6. Experimentation on the effectiveness of a few large functional programs with sets using the finite differencing optimization and the destructive update optimization.

In this chapter, we described the motivation, the background, and the project plan of this dissertation. This dissertation contributes in the following aspects:

- We enrich the typed λ-calculus with set notations. Thus, we validate the enriching of all functional languages with set notations since all functional languages can be viewed as variations of λ-calculus [Yun97].
- We propose a new approach to destructive update analysis for the optimization of aggregates. We develop the semantic model of dynamic destructive

update analysis and the abstract interpretation of static destructive update analysis.

- We develop a framework for applying finite differencing together with the destructive update analysis to the optimization of functional programs with sets.

Chapter 2

Destructive Effect Analysis For Strict Functional Languages

In functional languages, the update operation typically requires copying the aggregate. The unnecessary copying of data can increase both the execution time and the memory requirements of an application [Blo89b, Hud87]. Therefore, a destructive update optimization for functional languages can make substantial improvement on the performance of the implementation of functional programs.

The cost of analyzing programs in order to allow destructive updates is recognized to be high. The classic static methods of compile-time destructive update analysis for lazy languages applying abstract interpretation frameworks over the abstract domains of $O(2^{2^N})$ size, where N is the length of the program [Blo94,

DP93, GH89, Hud87, Ode91]. The dynamic methods of run-time destructive optimization applied three-phase $O(N^4)$ algorithms for strict imperative languages with pointers [CBC93]. The high cost of both the compile-time analysis and the run-time optimization is due to the expensive maintenance of the equivalent relations or evaluation orders for the variables that share the same values when analyzing the execution of programs.

We propose a new approach, *destructive effect analysis*, to destructive update optimization for strict functional languages. In stead of maintaining the equivalent relations or evaluation orders, our analysis allows that the variables sharing the same value in a function in the standard semantics share a copy of same destructive effect. The dynamic version of our method takes $O(N^3)$ time to analyze a program. The static version of destructive update analysis applies an abstract interpretation framework over an abstract domain of size $O(2^N)$.

2.1 Introduction

Most implementations of functional languages employ at some level the notion of sharing, whether manifested indirectly through an environment or directly via pointers. Sharing is essential to an efficient implementation of a functional language in two aspects: saving time by avoiding value re-computation and saving space by keeping only one copy of each value [Hud87]. Although sharing is perva-

sive at the implementation level, it is rarely explicitly expressed in formal context. This causes a problem in performing a variety of useful optimizations at compile time with the information of sharing properties. Perhaps the most important of these arises in destructive update optimization for the languages supporting aggregate data structures.

Destructive update optimization can be described as follows:

Definition (Destructive Update Optimization): The destructive update optimization produces the following results:

1. The expression that binds a compound value V to a variable x is implemented by making x a pointer to V.
2. An updating expression is analyzed so that it will be implemented by a destructive update to the value if none of the variables pointing to the value will be referenced later in the program. Otherwise, the updating expression will be implemented by making a fresh copy of the value. □

Destructive update analysis reduces the implicit copies in the implementation of programs. It is also known as *copy elimination* optimization. The determination of whether destructive updates can be performed rely on the analysis whether all the variables bound to the value are dead. Therefore, destructive update analysis is closely related to alias analysis and liveness analysis. Moreover, since the variables

in different functions may bind to the same value, usually it involves interprocedural flow analysis as well. In the example of the following program segment, in the execution of function call $f\ Y\ b$, the variable S will be analyzed together with the variables pointing to the same value outside function f, namely X and Y.

```
1) fun f S a = S less a;
2) let Y = X;
3)  in f Y b;
```

The destructive update problem has been studied by two major groups of people that gave the solutions in two different ways. The group of people in imperative languages applied polynomial-time algorithms of a dynamic destructive update analysis. The dynamic solutions analyze programs at compile time and leave the final determination to run time according to the run-time information. The best known algorithm takes $O(N^4)$ time overall [CBC93]. Goyal and Paige propose a new algorithm of $O(N^3)$ in their recent paper [GP98]. The other group of people in lazy functional languages apply abstract interpretation frameworks over abstract domains of size $O(2^{2^N})$ or larger [Blo89a, Blo89b, Blo94, DP93, Ode91]. The static solutions examine all the possible control paths and all the possible values of function calls at compile time. A static analysis with an abstract interpretation framework requires a computation of the sets of all possible values over the powerdomains.

In this paper we present a precise semantic model of *destructive effect analysis* for the destructive update optimization of first-order functional languages. The algorithm of our dynamic destructive effect analysis takes $O(N^3)$ time to analyze a program. We also present an abstract interpretation framework of destructive effect analysis over a finite domain. The abstract interpretation framework has an abstract domain of size $O(2^N)$. We show how our method analyzes some nontrivial programs in an effective way.

The methodology that we use to model the destructive effect analysis is interesting in its own right. Instead of maintaining the equivalent relations or evaluation orders as the conventional approaches did, our approach analyzes on the same copy of the destructive effect for the variables sharing the same value in a function in the standard semantics. Also, our method combines a few ideas from different approaches of a destructive update optimization. Our analysis models the destructive effect of values in a way similar to the way that Hudak models reference counting of values [Hud87]. The time stamps that our analysis uses for analyzing the destructive effects is similar to the definition of the time stamps described in Wand and Clinger's model of the set constraints for a destructive update optimization of arrays [WC98].

This chapter is organized as follows. We present a time-stamped first-order functional language in the following section. In Section 2.3, we include the prelim-

inary ideas behind the destructive effect analysis. Section 2.4 presents the semantic model of the dynamic destructive effect analysis. Section 2.5 describes the abstract interpretation framework of destructive effect analysis. We discuss the issues of safety and complexity in Section 2.6. Section 2.7 is a demonstration of both the dynamic version and the abstract interpretation of destructive effect analysis on a practical bubble sort example. We discuss the related work in Section 2.8. We conclude this chapter by a brief conclusion and an appendix of the semantic model of time stamps.

2.2 Source Language

Our source language takes the form of mutually recursive first-order recursion equations with constants. The expressions in a program are time stamped with a lexicographical order. A semantic model for time stamping programs is included in the appendix of this chapter. In the source language, we model all objects, including compound objects and integers, in a uniform way.

Note that we assume that all nested lambda abstractions have been lifted to the top level [Joh85], and that we model the destructive effects through these top-level functions.

The abstract syntax of the source language is given as follows.

Abstract Syntax

$c \in Con$	constants
$x \in Bv$	bound variables
$up \in Upf$	primitive functions that update arguments
$np \in Npf$	primitive functions that do not update arguments
$f \in Fv$	function variable
$\theta \in \Theta$	lexicographically ordered time stamps
$e \in Exp$	expressions, where
	$e = c \mid x \mid up\ x \mid np\ x \mid f\ e_1\ \ldots\ e_n \mid \mathsf{if}\ e_0\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 \mid \mathsf{let}\ x = e_0\ \mathsf{in}\ e_1$
$pr \in Pr$	programs, where $pr = \{f_i(x_1, \ldots, x_n) = e_i\}$

In our abstract syntax, we distinguish the updating primitive functions Upf from the non-updating primitive functions Npf. For examples, $\mathsf{sel}(S,\ i) \in Npf$ and $\mathsf{upd}(S, i, x) \in Upf$.

Semantic Domains

Int	the standard domain of integers
$Bool$	the standard domain of boolean values
$D = Int + Bool + \{\bot\}$	the domain of basic values
$Fun = Bas^n \to Bas$	the domain of first-order functions
$Bve = Bv \to D$	the domain of bound variable environments
$Fve = Fv \to Fun$	the domain of function environments

We adopt the following conventional notations. Double brackets surround syntactic objects, as in $\mathcal{E}[[exp]]$. Square bracket are used for environment updates, as in $env[e/x]$. We use $[y_j/x_j]_1^n$ as the shorthand for $[y_1/x_1, \ldots, y_n/x_n]$.

Semantic Functions

$$\mathcal{E}_s : [[\Theta : Exp]] \to Bve \to Fve \to D$$

$$\begin{aligned}
\mathcal{E}_s\ [[\theta : c]]\ bve\ fve &= c,\ \text{constant} \\
\mathcal{E}_s\ [[\theta : x]]\ bve\ fve &= bve(x) \\
\mathcal{E}_s\ [[\theta : up\ x]]\ bve\ fve &= (\mathcal{E}_s\ [[up]]\ bve\ fve)\ (\mathcal{E}_s\ [[\theta : x]]\ bve\ fve) \\
\mathcal{E}_s\ [[\theta : np\ x]]\ bve\ fve &= \mathsf{np}\ (\mathcal{E}_s\ [[\theta : x]]\ bve\ fve) \\
\mathcal{E}_s\ [[\theta : \mathsf{let}\ x = e_0\ \mathsf{in}\ e_1]]\ bve\ fve &= \mathcal{E}_s[[\theta_1 : e_1]]\ bve[(\mathcal{E}_s\ [[\theta_0 : e_0]]\ bve\ fve)/x] \\
&\qquad\qquad bve \setminus \{x\}
\end{aligned}$$

$$
\begin{aligned}
\mathcal{E}_s\ [[\theta : f\ e_1\ \ldots\ e_n]]\ bve\ fve &= \mathsf{let}\ d_1 = \mathcal{E}_s\ [[\theta_1 : e_1]]\ bve\ fve \\
&\quad\ \vdots \\
&\quad\ d_n = \mathcal{E}_s\ [[\theta_n : e_n]]\ bve\ fve \\
&\quad\ \mathsf{in}\ fve(f)(d_1, \ldots, d_n) \\
\mathcal{E}_s\ [[\theta : \mathsf{if}\ e_0\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2]]\ bve\ fve &= (\mathcal{E}_s\ [[\theta_0 : e_0]]\ bve\ fve) \rightarrow \\
&\quad\ (\mathcal{E}_s\ [[\theta_1 : e_1]]\ bve\ fve), \\
&\quad\ (\mathcal{E}_s\ [[\theta_2 : e_2]]\ bve\ fve)
\end{aligned}
$$

The program semantic function $\mathcal{A}_s$ is defined as follows.

$$
\begin{aligned}
\mathcal{A}_s\ [[pr]] = \mathcal{A}_s\ [[\{f_i(x_1, \ldots, x_n)\}]] &= fve, \text{ where} \\
fve &= [(\lambda(y_1, \ldots, y_n).\mathcal{E}_s\ [[\theta_i : e_i]]\ [y_j/x_j]_1^n)/f_i]_1^n
\end{aligned}
$$

For simplicity we assume that the first function f_1 takes no arguments, and a program is executed by calling f_1. More specifically, to execute a program $pr = \{f_i(x) = e_i\}$ is to evaluate $\mathcal{A}_s\ [[pr]]\ [[f_1]]$.

2.3 Preliminaries

There are some notions the understanding of which helps clarify the semantic model of destructive effect analysis.

2.3.1 Liveness Analysis with Time Stamps

We extend the idea of using lexicographically ordered time stamps for program analysis described in [WC98]. Wand and Clinger applied a time stamp scheme for a language with a continuation passing style semantics. The time stamps are used for deciding the order of the executions of expressions.

In the discussion of this chapter, we characterize a time stamp scheme by a function $\mathcal{C}(\theta_1, \theta_2)$ for comparing two time stamps θ_1 and θ_2. The functionality of $\mathcal{C}$ is defined as follows.

$$\mathcal{C}(\theta_1,\ \theta_2) = \begin{cases} -1 & \text{if } \theta_1 \text{ happens earlier than } \theta_2\ (\theta_1 < \theta_2) \\ 1 & \text{if } \theta_1 \text{ happens later than } \theta_2\ (\theta_1 > \theta_2) \\ 0 & \text{if } \theta_1 = \theta_2 \\ 2 & \text{if } \theta_1 \text{ and } \theta_2 \text{ are in different arms of an if expression} \\ & (\theta_1 \cong \theta_2) \end{cases}$$

A detailed algorithm for computing $\mathcal{C}$ is included in the appendix.

With time stamps, we may describe a variable liveness analysis as follows.

> Definition (Variable Liveness Analysis with Time Stamps): We say x is live at θ if and only if $\exists\theta_i < \theta$ such that x is initialized at θ_i, and $\exists\theta_a > \theta$ such that x is referenced at θ_a. $\square$

Let Θ be a set of time stamps. The minimum of a set of time stamps, $min\Theta$,

is defined as follows.

$$min\ \Theta = \{\theta \in \Theta | \forall \theta' \in \Theta,\ \mathcal{C}(\theta, \theta') \neq 1\}.\Box$$

It is clear that a minimum time stamp set has the following property.

$$\forall \theta_1, \theta_2 \in min\Theta,\ \text{such that } \theta_1 \cong \theta_2.\Box$$

We may analyze the liveness of a value with the liveness of the variables that the value is bound to.

Definition (Value Liveness Analysis with Time Stamps): Consider a value V that is bound to variables $x_1, \ldots, x_k$, which are initialized at $\theta_{i_1}, \ldots, \theta_{i_k}$, respectively. We say that V is initialized at $\Theta_{i_V} = min\{\theta_{i_1}, \ldots, \theta_{i_k}\}$. V is live at θ if $\exists \theta_{iV} \in \Theta_{iV}$ such that $\theta > \theta_{i_V}$ and $\exists \theta_a > \theta$, $\exists x \in \{x_1, \ldots, x_k\}$ such that x is referred at θ_a. $\Box$

For an example, in the following program segment, the variable x is dead after line 1 while the value bound to x is live until the end of the program execution.

```
1.      y = x;
2.0     if i > j
2.1        then z = upd (y, i, 0);
2.2        else z = y;
```

2.3.2 Variable Last-Use Analysis

For a destructive update analysis, we want to answer the question: "For the expression θ : up x, is the value bound to x referred at θ for the last time in the execution of the program?" This leads to the computation of *last-use*.

We define the last-use of a variable as follows.

> Definition (Last-Use of a Variable): The last-use of a variable x, $l(x)$ is a set of time stamps such that $\forall\theta_l \in l(x)$ the expression $\theta_l : e_l$ refers to x, and that $\forall\theta_a > \theta_l$ there is no reference to x at θ_a. □

When analyzing the expression $\theta : e$ that refers to x with a last-use $l(x)$, we use an auxiliary function $\mathcal{S}(l(x),\ \theta)$ to compute the new last-use of x after analyzing the expression. The definition of $\mathcal{S}$ is as follows.

$$\mathcal{S}(l(x),\ \theta) = \begin{cases} \{\theta_x \in l(x) \mid \mathcal{C}(\theta_x,\ \theta) > -1\} & \text{if } \exists\theta_x \in l(x),\ \mathcal{C}(\theta_x,\ \theta) = 1 \\ \{\theta_x \in l(x) \mid \mathcal{C}(\theta_x,\ \theta) > -1\} \cup \{\theta\} & \text{otherwise.} \end{cases}$$

It is clear that $\forall\theta_1 \in l(x)$ such that $\theta_1 < \theta$, θ_1 is removed from $l(x)$, and that if there is no $\theta_2 \in l(x)$ such that $\theta_2 > \theta$, θ is inserted into $l(x)$.

We present the semantic model of the variable last-use analysis in Figure 2.1. Please note that the variable last-use environment *lbve* is computable at compile time since it does not rely on any run-time information. The time required for the variable last-use analysis is $O(N \times s \times d)$, where s is the maximum number

Semantic Domains

$$
\begin{array}{rcl}
\Theta & = & \text{the domain of lexicographically ordered time-stamps} \\
Lbve & = & Bv \to 2^{\Theta} \\
Lfve & = & Fv \to Lbve
\end{array}
$$

Semantic Functions

$$\mathcal{E}_l : [[\Theta : Exp]] \to Lbve \to Lbve$$

$$
\begin{array}{rcl}
\mathcal{E}_l\ [[\theta : c]]\ lbve & = & lbve,\ \text{constant } c \\
\mathcal{E}_l\ [[\theta : x]]\ lbve & = & lbve[\mathcal{S}(\Theta,\ \theta)/x],\ \text{where } \Theta = lbve(x) \\
\mathcal{E}_l\ [[\theta : up\ x]]\ lbve & = & \mathcal{E}_l\ [[\theta : x]]\ lbve \\
\mathcal{E}_l\ [[\theta : np\ x]]\ lbve & = & \mathcal{E}_l\ [[\theta : x]]\ lbve \\
\mathcal{E}_l\ [[\theta : f\ e_1\ \ldots\ e_n]]\ lbve & = & \mathcal{E}_l\ [[\theta_n : e_n]]\ (\mathcal{E}_l\ [[\theta_{n-1} : e_{n-1}]]\ (\ldots \\
 & & \quad (\mathcal{E}_l\ [[\theta_1 : e_1]]\ lbve)\ldots)) \\
\mathcal{E}_l\ [[\theta : \mathsf{let}\ x = e_0\ \mathsf{in}\ e_1]]\ lbve & = & \mathcal{E}_l\ [[\theta_1 : e_1]]\ (\mathcal{E}_l\ [[\theta_0 : e_0]]\ lbve)[\mathcal{S}(\Theta,\ \theta_0)/x] \\
 & & \text{where } \Theta = lbve(x) \\
\mathcal{E}_l\ [[\theta : \mathsf{if}\ e_0\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2]]\ lbve & = & \mathcal{E}_l\ [[\theta_2 : e_2]]\ (\mathcal{E}_l\ [[\theta_1 : e_1]] \\
 & & \quad (\mathcal{E}_l\ [[\theta_0 : e_0]]\ lbve))
\end{array}
$$

$$
\begin{array}{rcl}
\mathcal{A}_l\ [[\{\theta : f_i(x_1, \ldots, x_n) = e_i\}]] & = & lfve \\
\text{where } lfve & = & [\mathcal{E}_l\ [[\theta_i : e_i]]\ []/f_i]_1^n
\end{array}
$$

Figure 2.1: The variable last-use analysis

of branches of nested if expressions and d is the maximum depth of nested if expressions.

Lemma 1 *By scanning a program once, we may collect the last-use information of all variables of a program in* $O(N \times s \times d)$ *time.*

Proof: The number of time stamps in the last-use of a variable x is $O(s)$. It takes $O(s \times d)$ time to update the last-use of x when x is referenced. And, therefore the total time for a variable last-use analysis is $O(N \times s \times d)$. □

It is clear that s and d are both $O(N)$ in the worst case. We say that the compile-time analysis for the dynamic destructive effect analysis takes $O(N^3)$ time to analyze a program.

The last-use of a value is defined as follows.

Definition (Last-Use of a Value): Consider a value V that is bound only to variables $x_1, \ldots, x_k$. The last-use of V, $l(V)$ is a set of time stamps such that $\forall \theta_l \in l(V)$, $\forall \theta_a > \theta_l$, there is no reference to any of $x_1, \ldots, x_k$ at θ_a. □

The last-use of a value V is included as a part of the *destructive effect* of V. We will describe more of it in Section 2.4.

2.3.3 Destructibility

In the cases of no function calls involved, the decision whether we may destructively update a value is straightforward. We call the decision as *local destructibility*. The local destructibility of a value in a function is defined as follows.

Definition (Local Destructibility):

For a value V in function f, the local destructibility of V at θ, $ld_f(V, \theta)$, is *true* if and only if $\exists \theta_v \in \Theta_{fV}$ such that $\theta_v = \theta$ or $\theta_v < \theta$, where Θ_{fV} is the last-use of V in f. $\Box$

For example, in the following program segment, the value bound to x is not locally destructible at line 1 while it is locally destructible at line 2.

```
1.   y = upd (x, 1, 0);
2.   z = upd (x, 2, 0);
```

Consider the following main function of a program.

```
1. f x;
2. g x;
```

The execution of function f can not destructively update the value bound to x in any case, while the execution of function g may destructively update x at the last reference to the value bound to x. We call this attribute of a value *interprocedural destructibility*.

Definition (Interprocedural Destructibility): Consider a value V in a function f.

1. If V is created in the execution of f, the interprocedural destructibility of V, id_{fV}, is $true$ through the execution of f.
2. When V is passed into f as a parameter at θ in function g, the interprocedural destructibility of V in f, id_{fV}, is $true$ if and only if
   - id_{gV} is $true$, and
   - $ld_g(V, \theta)$ is $true$. $\square$

The variables in different functions may refer to the same value by passing the value as a function argument. In a single-threaded lambda-lifted program, the global destructibility of a value V is defined as follows.

Definition (Global Destructibility): Consider a value V in a function f. The global destructibility of V at time stamp θ in function f, $gd_f(V, \theta)$, is $true$ if and only if

- id_{fV} is $true$, and
- $ld_f(V, \theta)$ is $true$. □

The global destructibility of a value is important for destructive effect analysis. We will describe more of it in the following section.

2.4 Formal Semantics of Destructive Effect Analysis

The goal of our destructive effect analysis is to facilitate a destructive update optimization in the following way.

Definition (Destructive Update Optimization with Destructive Effect Analysis): A destructive update optimization with the destructive effect analysis implements an updating primitive on a value V at time stamp θ with a destructive update iff the global destructibility of V, $gd_f(V, \theta)$ is $true$. □

We define the destructive effect μ of a value V in function f as

$$\mu_f(V) = \langle des,\ \Theta \rangle$$

where, des is the interprocedural destructibility of V passed into f when f is called, and Θ is the last-use of V in f.

For example, a fresh value V in function f has the destructive effect $\mu_f(V) = \langle true,\ \emptyset \rangle$. Note that $des(V) = true$ indicates the fact that V may be destructively

updated at the last reference in f.

Now we describe the semantic model of the dynamic destructive effect analysis.

Semantic Domains

Θ	the domain of lexicographically ordered time-stamps
Φ	the domain of locations
$\mathcal{U} = Bool \times 2^{\Theta}$	the domain of destructive effects
$Fun =$	$\Phi^n \rightarrow Int$, $n \in Int$
$Bve =$	$Bv \rightarrow \Phi$
$Fve =$	$Fv \rightarrow Fun$
$\Psi = \Phi \rightarrow \mathcal{U}$	stores
$Lbve =$	$Bv \rightarrow 2^{\Theta}$
$Lfve =$	$Fv \rightarrow Lbve$
$lfve \in Lfve$	the last-use information from the variable last-use analysis

The destructive effects are modeled with a store Φ and a store ψ ($\psi \in \Psi$) is a map from locations to destructive effects. A bound variable environment *bve* maps bound variables to the locations in the store where the destructive effects of the values are stored. A modeled function fve gives the meaning of a lambda-lifted top-level functions of the program, analogous to the fve in the standard semantics.

Our destructive effect analysis works with the time stamp schemes that systematically denote the program points so that the time stamps may be compared.

In other words, our destructive effect analysis works with any time stamp schemes in which the comparing function $\mathcal{C}$ is defined. A detailed time stamp scheme is described in the appendix of this chapter.

We present the semantic functions of the dynamic destructive effect analysis in Figures 2.2. The operational semantics that we capture may be summarized as follow. A program begins with an empty destructive effect store. As the program executes, the expressions modify the destructive effects of values. When a value V is bound to a variable x, the last-use of V is updated with the auxiliary function $\mathcal{L}$. Upon function calls, the interprocedural destructibilities of the arguments are evaluated with the auxiliary function $\mathcal{D}$ before passing the values into the functions. Hence, for the updating primitives, the destructive update optimization can be performed when the global destructibility of the value is $true$.

When V, with last-use Θ_V, is bound to a variable x, with last-use Θ_x, we use the auxiliary function $\mathcal{L}(\Theta_V,\ \Theta_x)$ to compute the last-use of V after binding.

$$\mathcal{L}(\Theta_V,\ \Theta_x) = \bigcup_{\theta_x \in \Theta_x} \mathcal{S}(\Theta_V,\ \theta_x),$$

where $\mathcal{S}$ is defined in section 2.3.2.

We use the auxiliary function $\mathcal{D}(\theta : \phi)$ to evaluate the global destructibility of a value stored at location ϕ.

$$\mathcal{D}(\theta : \phi) = \begin{cases} true & \text{if } des = true \text{ and } \exists \theta_V \in \Theta.\ \mathcal{C}(\theta, \theta_V) = 0 \text{ or } 1 \\ false & \text{otherwise,} \end{cases}$$

Semantic Functions

$$\mathcal{E}_d : [[\Theta : Exp]] \to Bve \to Fve \to \Psi \to Lbve \to \Phi$$

$$\begin{array}{rcl}
\mathcal{E}_d\ [[\theta : c]]\ bve\ fve\ \psi\ lbve & = & \phi,\ \psi[\langle true,\ \emptyset\rangle/\phi] \\
\mathcal{E}_d\ [[\theta : x]]\ bve\ fve\ \psi\ lbve & = & bve(x) \\
\mathcal{E}_d\ [[\theta : up\ x]]\ bve\ fve\ \psi\ lbve & = & \left\{ \begin{array}{ll} \phi_x,\ \psi[\langle true,\ \emptyset\rangle/\phi_x] & \text{if } \mathcal{D}(\theta : bve(x)) \\ \phi',\ \psi[\langle true,\ \emptyset\rangle/\phi'] & \text{otherwise} \end{array} \right. \\
\mathcal{E}_d\ [[\theta : np\ x]]\ bve\ fve\ \psi\ lbve & = & \mathsf{np}\ (\mathcal{E}_d\ [[\theta : x]]\ bve\ fve\ \psi\ lbve)
\end{array}$$

$$\begin{array}{rcl}
\mathcal{E}_d\ [[\theta : f\ e_1\ \ldots\ e_n]]\ bve\ fve\ \psi\ lbve & = & \mathsf{let}\ \phi_1 = \mathcal{E}_d\ [[\theta_1 : e_1]]\ bve\ fve\ \psi\ lbve \\
 & & \vdots \\
 & & \phi_n = \mathcal{E}_d\ [[\theta_n : e_n]]\ bve\ fve\ \psi\ lbve \\
 & & \mathsf{in}\ \mathcal{R}_d(fve(f).(\mathcal{P}(\theta_n : \phi_1), \ldots, \mathcal{P}(\theta_n : \phi_n))) \\
\mathcal{E}_d\ [[\theta : \mathsf{let}\ x = e_0\ \mathsf{in}\ e_1]]\ bve\ fve\ \psi\ lbve & = & \mathcal{E}_d[[\theta_1 : e_1]]\ bve\ fve\ \psi\ lbve,\ bve \setminus \{x\} \\
 & & \text{where,} \phi_u = \mathcal{E}_d\ [[\theta_0 : e_0]]\ bve\ fve\ \psi\ lbve, \\
 & & \Theta_x = lbve(x),\ \langle des,\ \Theta\rangle = \psi(\phi_u), \\
 & & \psi[\langle des, \mathcal{L}(\Theta,\ \Theta_x)\rangle/\phi_u], \text{and } bve[\phi_u/x]
\end{array}$$

$$\begin{array}{rcl}
\mathcal{E}_d\ [[\theta : \mathsf{if}\ e_0\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2]]\ bve\ fve\ \psi\ lbve & = & (\mathcal{E}_d\ [[\theta_0 : e_0]]\ bve\ fve\ \psi\ lbve) \to \\
 & & (\mathcal{E}_d\ [[\theta_1 : e_1]]\ bve\ fve\ \psi\ lbve), \\
 & & (\mathcal{E}_d\ [[\theta_2 : e_2]]\ bve\ fve\ \psi\ lbve)
\end{array}$$

$$\begin{array}{rcl}
\mathcal{A}_d\ [[\{f_i(x_1, \ldots, x_n) = \theta_i : e_i\}]]\ lfve & = & fve, \text{ where} \\
fve & = & [(\lambda(y_1, \ldots, y_n).\ fun_i\ bve_i\ lbve_i)/f_i]_1^n \\
\text{where} & & bve_i = [y_j/x_j]_1^n, lbve_i = lfve(f_i), \text{ and} \\
fun_i & = & \mathsf{let}\ \langle des_1,\ \Theta_1\rangle = \psi(y_1), \\
 & & \quad \psi[\langle des_1,\ \mathcal{L}(\Theta_1,\ lbve_i(x_1))\rangle/y_1], \\
 & & \quad \vdots \\
 & & \quad \langle des_n,\ \Theta_n\rangle = \psi(y_n), \\
 & & \quad \psi[\langle des_n,\ \mathcal{L}(\Theta_n,\ lbve_i(x_n))\rangle/y_n] \\
 & & \mathsf{in}\ \mathcal{Q}(\mathcal{E}_d\ [[\theta_i : e_i]]\ bve_i\ fve\ \psi\ lbve_i)
\end{array}$$

Figure 2.2: Semantic functions of dynamic destructive effect analysis

where $\langle des,\ \Theta \rangle = \psi(\phi)$.

In destructive effect analysis, the functions are modeled in the following way.

- A function takes the interprocedural destructibility of each argument as a part of the parameter. Based on the interprocedural destructibility, we compute the global destructibility of the value and analyze whether the value can be destructively updated in the execution of the function.

- A function either returns a fresh value created in the execution of the function or returns one of its parameters. Since all the fresh values have the same destructive effect value, we use an auxiliary function $\mathcal{Q}$ to encode the return values into integers, where 0 indicates that the return value is created in the execution of the function, and for $1 \leq i \leq n$, i indicates that the return value is the i^{th} parameter passed into the function.

$$\mathcal{Q}(\phi) = \begin{cases} i & \text{if } \phi \text{ is the } i^{th} \text{ argument, } 1 \leq i \leq n \\ 0 & \text{otherwise.} \end{cases}$$

Upon the function calls, each variable that appears in the arguments are considered as a reference no matter whether the value is referred in the execution of the function body or not. This is analogous to the standard semantics of strict languages in which all arguments are evaluated before passing into functions. Note that some of the popular approaches also analyze programs in a similar way, such as [Hud87].

Following the way that functions are modeled, function calls are modeled as follows. For a function call $f\ e_1\ \ldots\ e_n$, the arguments are evaluated into $\phi_1, \ldots, \phi_n$, respectively.

- When passing a value V at location ϕ to a function f at θ, we initialize a new copy of destructive effect for the analysis of V in the execution of f. The new destructive effect is initialized as $\langle \mathcal{D}(\phi), \emptyset \rangle$, and passed into f as a formal parameter.

- For the arguments pointing to the same value in a function call, the same locations of destructive effect are passed into the function. In the following example, the function f gets the same destructive effect location for the both arguments in the function call.

```
let y = x;
 in f (x, y);
```

  The auxiliary function $\mathcal{P}$ maps the arguments of the same value to the same location of destructive effect so that the formal parameters of the same value will be analyzed on the same copy of the destructive effect when analyzing

the execution of the function.

$$\mathcal{P}(\theta : \phi) = \begin{cases} \phi',\ \psi[\langle \mathcal{D}(\theta : \phi),\ \emptyset \rangle / \phi'] & \text{if } \phi \in \{\phi_1, \ldots, \phi_n\} \\ \bot & \text{otherwise} \end{cases}$$

- After a function call is evaluated, we use the auxiliary function $\mathcal{R}_d$ to translate a encoded return value into the corresponding destructive effect.

$$\mathcal{R}_d(k) = \begin{cases} \phi_k & \text{if } 1 \leq k \leq n \\ \phi',\ \psi[\langle true,\ \emptyset \rangle / \phi'] & \text{otherwise} \end{cases}$$

  If the return value is k, $1 \leq k \leq n$, the function actually returns the k^{th} argument, and the destructive effect analysis on the return value will be performed on the destructive effect of the k^{th} argument passed into the function for the analysis of the rest of the program execution. Otherwise, the return value is 0, meaning that the return value is initialized in the execution of f, and therefore we initialize the destructive effect as $\langle true, \emptyset \rangle$ for the analysis on the return value for the rest of the program execution.

Figure 2.3 demonstrates the destructive effect operations. Suppose that at some snapshot of a program execution, there are k active functions. For a value V, we have k copies of destructive effects of V, each of which is for the analysis of an active function. All the variables in a function that point to V share the same copy of destructive effect.

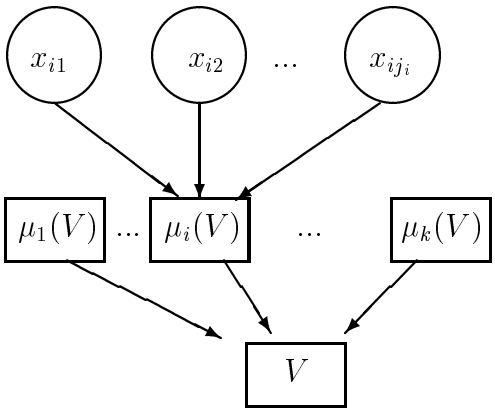


Figure 2.3: The destructive effects of a value

For example, consider the following simple program that initializes the elements of an array of 100 integers to 0's.

```
        fun init ( x, i ) =
1.          if i < 1
1(1).1.         then x;
1(2).1.         else let y = upd(x, i, 0);
1(2).2.                in init(y, i-1);


1.      init(a, 100);
```

The last-use of a is $\{1\}$ and the interprocedural destructibility of a is *true* when passing a into *init*. The last-use of x in *init* is $\{1(1).1, 1(2).1\}$ so that the `upd`

primitive at 1(2).1 is implemented by a destructive update. The last-use of y in *init* is $\{1(2).2\}$ so that the interprocedural destructibility of y is *true* when passing into the recursive *init*. Hence, all the 100 upd primitives along the execution of the program will all be implemented by destructive updates according to our destructive update optimization with destructive effect analysis.

Theorem 2. *The dynamic destructive effect analysis is safe.*

Proof: In the dynamic destructive effect analysis, a destructive update is performed on the value V for an updating primitive at time stamp θ in function f only if the global destructibility V, $gd_f(V, \theta)$, is *true*. By the definition of global destructibility,

- $ld_f(V, \theta)$ is *true*, and
- id_{fV} is also *true*.

$ld_f(V, \theta)$ is *true* implies that V will not be referenced in the rest of function f. id_{fV} is *true* implies that for the function g calling to f at θ_g, $gd_g(V, \theta_g)$ is *true*.

We show that when $gd_f(V, \theta)$ is *true*, V will not be referenced later in the execution of the program by an induction on the depth of the calling path, i. Let f_1 be the main function.

- Base case $i = 1$: That is, θ occurs in f_1, the main function.

$gd_{f_1}(V, \theta)$

$\Leftrightarrow$ $ld_{f_1}(V, \theta) = true$ and $id_{f_1 V} = true$

$\Leftrightarrow$ There is no reference to V later in f_1 and

V is created in f_1

$\Leftrightarrow$ V will not be reference later in the execution

of the program

- Induction: Suppose $i = k$ $(1 \leq k)$ it is true that when $gd_{f_k}(V, \theta) = true$, V will not be referenced later in the execution of the program.

  $gd_{f_{k+1}}(V, \theta)$

  $\Leftrightarrow$ $ld_{f_{k+1}}(V, \theta) = true$ and $id_{f_{k+1} V} = true$

  $\Leftrightarrow$ There is no reference to V later in f_{k+1} and

  V is created in f_1 or $gd_{f_k}(V, \theta_{f_k}) = true$

  $\Leftrightarrow$ V will not be reference later in the execution

  of the program

This completes the proof that if $gd_f(V, \theta)$ is $true$, V will not be referred to in the rest of the execution of the program.

Hence, we proved that when the dynamic destructive update optimization using the dynamic destructive effect analysis implements an updaing expression by a destructive update, there is no reference to the value later in the execution of the program; that is, the dynamice effect

analysis is safe. □

The dynamic destructive effect analysis consists of two phases:

- a variable last-use analysis at compile time, and
- computing destructive effects at run time.

We proved that for a program, a variable last-use analysis takes $O(N^3)$ time. Now, we analyze the time required for computing a destructive effect as follows.

Lemma 3. *In the dynamic destructive effect analysis, computing the destructive effect of an expression introduces an $O(s \times d)$ overhead of execution time, where N is the length of the program, s is the maximum number of branches of nested* if *expressions and d is the maximum depth of nested* if *expressions.*

Proof: It follows from the facts that it takes $O(d)$ time for each time stamp comparison and that there are at most $O(s)$ time stamps in Θ_V, the last-use of V. □

It is clear that both s and d are $O(N)$ in the worst case. We say that the dynamic destructive effect analysis introduces an $O(N^2)$ overhead to each expression for computing the destructive effect at run time.

2.5 Abstract Interpretation of Destructive Effect Analysis

The semantic model of dynamic destructive effect analysis presented in Figure 2.2 is exact, and thus evaluating a destructive effect may not terminate any more than a program in the standard semantics would. For an analysis at compile time we may abstract the destructive effect information away from the standard semantics.

Without the standard information of a predicate the destructive effect of an if expression must be approximated with the assumption that either arm of an if expression could be taken. This initially leads to two possible destructive effects through each conditional, and ultimately to a set of possible destructive effects through the program. Moreover, the destructive effect of a function applications is approximated with the destructive effects of all possible values that the functions may possibly return.

Semantic Domains

Θ	the domain of lexicographically ordered time-stamps
Φ	the domain of locations
$\mathcal{U} = Bool \times 2^{\Theta}$	the domain of destructive effects
$Fun =$	$\Phi^n \rightarrow 2^{Int}$, $n \in Int$
$Bve =$	$Bv \rightarrow \Phi$
$Fve =$	$Fv \rightarrow \Phi$
$\Psi = \Phi \rightarrow \mathcal{U}$	stores
$Lbve =$	$Bv \rightarrow 2^{\Theta}$
$Lfve =$	$Fv \rightarrow Lbve$
$lfve \in Lfve$	the last-use information from the variable last-use analysis

In the semantic functions of the destructive effect abstract interpretation presented in Figure 2.4, the auxiliary function $\mathcal{P}$ described in the previous section is used for function applications, and the auxiliary function $\mathcal{R}_a$ is used for translating the set of return values into the corresponding set of destructive effects.

$$\begin{aligned}\mathcal{R}_a(fun,\ y_1, \ldots, y_n) \ &= \ \{y_i \mid i \in fun(\mathcal{P}(\theta_{:}y_1), \ldots, \mathcal{P}(\theta_n : y_n))\} \cup \\ &\qquad \{\phi, \psi[\langle true,\ \emptyset\rangle/\phi] \mid 0 \in fun(\mathcal{P}(\theta_1 : y_1), \ldots, \mathcal{P}(\theta_n : y_n))\}\end{aligned}$$

We summarize the operations of the destructive effect abstract interpretation as follows. The analysis starts with an empty set of destructive effect store. As the analysis performs, the set of the possible destructive effects of each expression

Semantic Functions

$$\mathcal{E}_a : [[\Theta : Exp]] \to Bve \to Fve \to \Psi \to Lbve \to 2^{\Phi}$$

$$\begin{array}{rcl}
\mathcal{E}_a\ [[\theta : c]]\ bve\ fve\ \psi\ lbve & = & \{\phi\},\ \psi[\langle true,\ \emptyset\rangle/\phi] \\
\mathcal{E}_a\ [[\theta : x]]\ bve\ fve\ \psi\ lbve & = & bve(x) \\
\mathcal{E}_a\ [[\theta : up\ x]]\ bve\ fve\ \psi\ lbve & = & \mathcal{H},\ \psi[\langle true,\ \emptyset\rangle/\phi \mid \phi \in \mathcal{H}] \\
\text{where } \mathcal{H} & = & \{\phi \in bve(x) \mid \mathcal{D}(\theta : \phi)\} \cup \\
 & & \quad \{\phi' \mid \exists \phi \in bve(x),\ \mathcal{D}(\theta : \phi) = false\} \\
\mathcal{E}_a\ [[\theta : np\ x]]\ bve\ fve\ \psi\ lbve & = & \{\mathsf{np}\ \phi \mid \phi \in bve(x)\} \\
\mathcal{E}_a\ [[\theta : f\ e_1\ \ldots\ e_n]]\ bve\ fve\ \psi\ lbve & = & \{\mathcal{R}_a(fun,\ y_1, \ldots, y_n) \mid fun = fve(f),\ \text{and} \\
 & & \quad \forall 0 \leq i \leq n, y_i \in (\mathcal{E}_a\ [[e_i]]\ bve\ fve\ \psi\ lbve)\}
\end{array}$$

$$\begin{array}{rcl}
\mathcal{E}_a\ [[\theta : \mathsf{let}\ x = e_0\ \mathsf{in}\ e_1]]\ bve\ fve\ \psi\ lbve & = & \mathcal{E}_a[[\theta_1 : e_1]]\ bve\ fve\ \psi\ lbve,\ bve \setminus \{x\} \\
\text{where} & & \Phi = \mathcal{E}_a\ [[\theta_0 : e_0]]\ bve\ fve\ \psi\ lbve, \\
 & & bve[\Phi/x],\ \Theta_x = lbve(x),\ \text{and} \\
 & & \psi[\langle des, \mathcal{L}(\Theta,\ \Theta_x)\rangle/\phi \mid \phi \in \Phi,\ \text{and} \\
 & & \quad \langle des,\ \Theta\rangle = \psi(\phi)] \\
\mathcal{E}_a\ [[\theta : \mathsf{if}\ e_0\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2]]\ bve\ fve\ \psi\ lbve & = & (\mathcal{E}_a\ [[\theta_0 : e_0]]\ bve\ fve\ \psi\ lbve, \\
 & & \quad \mathcal{E}_a\ [[\theta_1 : e_1]]\ bve\ fve\ \psi\ lbve) \cup \\
 & & (\mathcal{E}_a\ [[\theta_0 : e_0]]\ bve\ fve\ \psi\ lbve, \\
 & & \quad \mathcal{E}_a\ [[\theta_2 : e_2]]\ bve\ fve\ \psi\ lbve)
\end{array}$$

$$\begin{array}{rcl}
\mathcal{A}_a\ [[\{f_i(x_1, \ldots, x_n) = \theta_i : e_i\}]]\ lfve & = & fve,\ \text{where} \\
fve & = & [(\lambda(y_1, \ldots, y_n).\ fun_i\ bve_i\ lbve_i)/f_i]_1^n \\
\text{where } fun_i = & \mathsf{let} & \langle des_1,\ \Theta_1\rangle = \psi(y_1), \\
 & & \psi[\langle des_1,\ \mathcal{L}(\Theta_1,\ lbve_i(x_1))\rangle/y_1], \\
 & & \vdots \\
 & & \langle des_n,\ \Theta_n\rangle = \psi(y_n) \\
 & & \psi[\langle des_n,\ \mathcal{L}(\Theta_n,\ lbve_i(x_n))\rangle/y_n], \\
 & & bve_i = [\{y_j\}/x_j]_1^n,\ lbve_i = lfve(f_i), \\
 & \mathsf{in} & \{\mathcal{Q}(\phi) \mid \phi \in (\mathcal{E}_a\ [[\theta_i : e_i]]\ bve_i\ fve\ \psi\ lbve_i)\}
\end{array}$$

Figure 2.4: Semantic Functions of destructive effect abstract interpretation

is computed. When a value is bound to a variable, the last-use of the value is updated according to the last-use of the variable. Upon function calls, the interprocedural destructibilities of arguments are evaluated with the auxiliary function $\mathcal{D}$ (presented in the previous section) before passing the values into the functions, and the destructive effect set of the return values is the union of the destructive effect sets for the function applications computed over all possible values of each argument. Hence, for the updating primitives, the destructive updates can be performed if the analysis of all the possible destructive effects show that the values are globally destructible.

Note that instead of computing over the powerdomains of the destructive effects, we compute the values of function applications over the powerdomains of the interprocedural destructibilities of the arguments in the abstract domain. This is important for analyzing the safety and complexity of the destructive effect abstract interpretation, and we will discuss more of this in the following section.

2.6 Safety And Complexity

In this section, we discuss the issues on the safety and the complexity of the destructive effect abstract interpretation.

We summarize the operation of the abstract interpretation framework on the functions in the abstract domain as follows. Our analysis translates a function

$f \in (D^n \rightarrow D)$ in the standard domain into $fun \in (\mathcal{U}^n \rightarrow 2^{\mathcal{N}})$ in the abstract domain, where $\mathcal{N}$ is the set $\{0..n\}$. The auxiliary function $\mathcal{Q}$ presented in Section 2.4 is used to translate the destructive effects into integers such that i indicates the return value is the i^{th} argument for $1 \leq i \leq n$, and 0 indicates the return value is initialized in the execution of the function. Two auxiliary functions $\mathcal{P}$ and $\mathcal{R}_a$ are used for evaluating function calls in the abstract domain. $\mathcal{P}$ initializes the destructive effects of the arguments in the function calls. $\mathcal{R}_a$ translates the integers returned from fun into the corresponding destructive effects. With $\mathcal{P}$ and $\mathcal{R}_a$, we analyze a function call $f\ e_1\ \ldots\ e_n$ in the standard semantics by evaluating $fun\ \mathcal{P}(e_1)\ \ldots\ \mathcal{P}(e_n)$ in the abstract domain. For a recursive function f, the evaluation of $f\ e_1\ \ldots\ e_n$ computes the fixed point of $fun\ \mathcal{P}(e_1)\ \ldots\ \mathcal{P}(e_n)$, written as $Fix(fun\ \mathcal{P}(e_1)\ \ldots\ \mathcal{P}(e_n))$.

Lemma 4. (Termination of $Fix(fun\ \mathcal{P}(e_1)\ \ldots\ \mathcal{P}(e_n))$): *For a finite program pr in the abstract domain, $Fix(fun\ \mathcal{P}(e_1)\ \ldots\ \mathcal{P}(e_n))$ is computable for all $f \in pr$ in the standard domain.*

Proof: $Fix(fun\ \mathcal{P}(e_1)\ \ldots\ \mathcal{P}(e_n))$ can be effectively computed in the standard iterative manner and the iteration terminates when the least upper bound of the chain is reached, which is guaranteed because the domains are finite and the chain is monotonically increasing. □

Corollary 5. (Computability of $\mathcal{E}_a\ [[f\ e_1\ \ldots\ e_n]]$): *For a finite program*

pr, $\mathcal{E}_a$ $[[f\ e_1\ \ldots\ e_n]]$ *bve* ψ *lbve is computable.*

Proof: This follows lemma 4 and the fact that there is no recursion in the evaluations of $\mathcal{P}$, $\mathcal{Q}$, and $\mathcal{R}_a$. □

Consider the following example program of initializing an array.

```
        fun init ( x, i ) =
1.          if i < 1
1(1).1.         then x;
1(2).1.         else let y = upd (x, i, 0);
1(2).2.                in init(y, i-1);


1.  let a = read();    // input an array at run-time
2.      j = read();    // input an integer at run-time
3.   in init(a, j);
```

The function *init* is analyzed as $\widehat{init}$ in the abstract domain.

$$\begin{aligned}
\widehat{init} &= Fix(\lambda\hat{x}.\lambda\hat{i}.(\{1\} \cup \mathcal{Q}(\mathcal{R}_a(\widehat{init}(true, true))))) \\
&= Fix(\lambda\hat{x}.\lambda\hat{i}.(\{1\} \cup (\hat{x} \rightarrow \{1\}, \{0\}))) \\
&= \lambda\hat{x}.\lambda\hat{i}.(\hat{x} \rightarrow \{1\}, \{0, 1\})
\end{aligned}$$

Hence, the analysis of the function call $init(a, j)$ evaluates $\mathcal{R}_a(\widehat{init}(true, true))$ in the abstract domain, and gets the result of $\mathcal{R}_a(\{1\}) = \{\phi_a\}$. That is, the set of possible return values has only one element, the destructive effect of the value bound to a. The analysis shows that with the destructive update optimization the function call $f(a,\ j)$ either returns the original value of a or returns a value which is destructively updated on a, and that in the both cases the analysis on the return value will be make on the destructive effect of the value passed into *init* as the first argument. □

Lemma 6. *If an* upd *expression is implemented with a destructive update by the static analysis, the* upd *expression will be implemented with a destructive update by the dynamic analysis.*

Proof: It follows from the following two facts. First, the destructive effect of an expression computed in the dynamic analysis is an element in the destructive effect set of the expression computed in the static analysis. And second, a destructive update made for a upd expression in the static analysis guarantees all the possible values in the destructive effect set are globally destructible. □

Note that the converse of Lemma 6 does not necessarily hold.

We are ready to show the safety of the destructive effect abstract interpretation.

Theorem 7. (Safety)*: $\mathcal{A}_a$ [[pr]] is computable for any finite program pr,*

and a destructive update optimization with destructive effect analysis is safe.

Proof: It follows from corollary 5 and lemma 6. □

Due to the high complexity of an abstract interpretation framework, it is difficult to analyze the complexity of the algorithms of abstract interpretation. For example, consider the complexity of computing the fixed point of $f_i\ x_1\ \ldots\ x_n$ for $1 \leq i \leq n$ in the abstract domain in the case that $f_1, \ldots, f_n$ are n mutually recursive functions. For the purpose of comparing the complexity of abstract interpretation frameworks, we usually consider the size of the powerdomains of the functions in the abstract domains since the computation for the function applications of recursive functions requires computing the fixed points of the abstract functions over the powerdomains.

Theorem 8. (Complexity)*: In the static destructive effect analysis, the powerdomain of the abstract functions has a size of* $O(2^N)$.

Proof: For the analysis of function applications, the way our analysis evaluates the functions in the abstract domain may be summarized as follows. After it is translated by the auxiliary function $\mathcal{P}$, the set of destructive effects passed into the abstract function has a size of 2 for each argument (the size of $\{true, false\}$). fun computes over the $O(2^N)$ combinations of the possible argument sets and generates a set of pos-

sible return values, which has a size of $O(2^N)$. And, $\mathcal{R}$ translates the set of return values into a set of possible destructive effects. It is clear that for the other cases, the order of the number of possible destructive effects for an expression follows that for function application. This completes the proof. □

2.7 Example

We use the functional bubble sort program in Figure 2.5 to demonstrate the dynamic destructive effect analysis and the destructive effect abstract interpretation.

The variable last-use analysis computes the last-use of variables as follows.

variable	Θ
a_1	$\{2\}$
a_2	$\{1(1).1,\ 1(2).1\}$
a_3	$\{1(1).1,\ 1(2).1(1).1,\ 1(2).1(2).1\}$
t	$\{3\}$
a	$\{1\}$
b_1	$\{4\}$
b_2	$\{2\}$
b_3	$\{3\}$
b_3	$\{5\}$

```
      swap(a1, i, j) =
1.        let k = sel(a1, i);
2.            t = upd(a1, i, sel(a1, j));
3.         in upd(t, j, k);

      bsort2(a2, n, c) =
1.        if c
1(1).1.        then bsort1(a2, n, 0, false);
1(2).1.        else a2;

      bsort1(a3, n, i, c) =
1.        if (i == n-1)
1(1).1.       then bsort2(a3, n-1, 0, c);
1(2).1.       else if (sel(a3, i) > sel(a3, i+1))
1(2).1(1).1.             then let t1 = swap(a3, i, i+1);
1(2).1(1).2.                  in bsort1(t1, n, i+1, true);
1(2).1(2).1.           else bsort1(a3, n, i+1, c);

      bubsort(a, n) =
1.        bsort2(a, n, true);

1.    let b1 = [3, 6, 9, 2, 4, 8, 1, 5, 7];
2.        b2 = swap(b1, 2, 3);
3.        b3 = bubsort(b1, 9);
4.        b4 = b1;
5.     in bubsort(b4, 9);
```

Figure 2.5: A functional bubble sort program

In the discussion we adopt the notation of labeling function calls. For example, the second invocation of function *swap* is denoted as $swap_2$.

2.7.1 Dynamic Destructive Effect Analysis

The dynamic destructive effect analysis performs a destructive update optimization along the execution of a program. We highlight some of the analysis on the bubble sort program in Figure 2.5 as follows.

- The binding of $V_1 = [3, 6, 9, 2, 4, 8, 1, 5, 7]$ to b_1 at time stamp 1 updates the destructive effect of V_1 as $\mu(V_1) = \langle true,\ \{4\}\rangle$.

- On passing parameters for function call $swap_2$ at time stamp 2, the argument a_1 gets $\mu_{swap_2}(V_1) = \langle false,\ \{2\}\rangle$. The upd expression at time stamp 3 in function *swap* will be implemented by making a fresh copy of V_1 since $des(V_1) = false$. Hence, $swap_2$ returns a fresh copy of destructive effect for the value V_2 initialized in the execution of *swap*. The binding to b_2 at time stamp 2 makes b_2 point to V_2, and the destructive effect of V_2, $\mu(V_2)$, is updated as $\langle true,\ \{2\}\rangle$.

- In a similar way, on passing parameters for function call $bubsort_1$ at time stamp 3, the argument a gets $\mu_{bubsort_1}(V_1) = \langle false,\ \{1.\}\rangle$. And therefore, a_2 of $bsort2$ and a_1 of *swap* get $\mu_{bsort2_2}(V_1) = \langle false,\ \{1(1).1,\ 1(2).1\}\rangle$ and $\mu_{swap_1}(V_1) = \langle false,\ \{2\}\rangle$, respectively. The upd expression in function *swap*

will be implemented by making a fresh copy of V_1 as a new value V_3. All the following updates in functions $bsort1$, $bsort2$, $swap$ are implemented by destructive updates since the value passed among the functions is V_3, and $des(V_3) = true$. Hence, the binding to b_3 at time stamp 3 makes b_3 point to V_3 which is initialized in the execution of $swap$, and the destructive effect of V_3 is updated as $\mu(V_3) = \langle true,\ \{3\}\rangle$.

- When V_1 is bound to b_4 at time stamp 4, the destructive effect of V_1, $\mu(V_1)$, is updated as $\langle true,\ \{5\}\rangle$.

- On passing parameters for function call $bubsort_2$ at time stamp 5, the argument a in $bubsort_2$ gets $\mu_{bubsort_2}(V_1) = \langle true,\ \{1\}\rangle$. Therefore, all the following `upd` expressions when $swap$ is called are implemented by destructive updates since $des(V_1) = true$.

2.7.2 Destructive Effect Abstraction Interpretation

The destructive effect abstract interpretation performs a destructive update optimization at compile-time as follows. Please note that a static destructive effect analysis does not rely on the value in the standard semantics, and therefore considers more cases than what will be actually executed. We highlight some of its operations on the bubble sort program in Figure 2.5 as follows.

- The computed functions in the abstract domain are shown as follows.

$$\begin{aligned}
\widehat{swap} &= \lambda\widehat{a1}.\lambda\hat{n}.(\widehat{a1} \to \{1\}, \{0\}) \\
\widehat{bsort1} &= \lambda\widehat{a3}.\lambda\hat{n}.(\widehat{a3} \to \{1\}, \{0,1\}) \\
\widehat{bsort2} &= \lambda\widehat{a2}.\lambda\hat{n}.(\widehat{a2} \to \{1\}, \{0,1\}) \\
\widehat{bubsort} &= \lambda\hat{a}.\lambda\hat{n}.(\hat{a} \to \{1\}, \{0,1\})
\end{aligned}$$

- The destructive effect sets of expressions at time stamps 1 and 2 have one element each, which are exactly the destructive effects computed in the dynamic destructive effect analysis. Therefore, the operations are similar to those of the dynamic destructive effect analysis.

- For the analysis of the function call $bubsort_1(b_1, 9)$ at time stamp 3, we compute the destructive effect of $bubsort_1(false, true)$ in the abstract domain, shown as follows.

$$\begin{aligned}
bubsort_1(false, true) &= \mathcal{R}_{a(bubsort)}(\widehat{bubsort}(false,\ true)) \\
&= \mathcal{R}_{a(bubsort)}(\{0,1\}) \\
&= \{\phi_{V_1}, \phi_{V_3}\}\ \square
\end{aligned}$$

  Hence, the binding to b_3 at time stamp 3 makes b_3 bind to a destructive effect set with two elements: one for the destructive effect of V_1 and the other for a fresh value V_3, initialized in the execution of function *swap*. Since without

the exact information in the standard semantics we analyze at compile time on both the consequences and the alternatives of if expressions.

- The binding expression at time stamp 4 binds b_4 to the destructive effect set $\{\phi_{V_1}\}$ and updates $\mu(V_1)$ as $\mu(V_1) = \langle true,\ \{5\}\rangle$.

- The analysis of function call $bubsort_2(b_4, 9)$ at time stamp 5 is different from the analysis of $bubsort_1$ at time stamp 3 in spite that the first arguments in both function calls get the value V_1. The analysis of $bubsort_2(b_4, 9)$ actually computes $bubsort_2(true, true)$ in the abstract domain, shown as follows.

$$\begin{aligned} bubsort_2(true, true) &= \mathcal{R}_{a(bubsort)}(\widehat{bubsort}(true,\ true)) \\ &= \mathcal{R}_{a(bubsort)}(\{1\}) \\ &= \{\phi_{V_1}\}\ \Box \end{aligned}$$

  That is, either $bubsort_2$ returns the original value of V_1 or $bubsort_2$ returns a value that is destructively updated on V_1. For both cases the returned destructive effect set contains only one element, $\mu(V_1)$.

2.8 Comparison

Since there are only few, if any, works on the static analysis for a destructive update optimization on strict languages, we compare our analysis with some of the well-known analyses, although the major goals of their analyses are different.

The works on a destructive update analysis for lazy languages show the most similarity to our work. We discuss three of the popular analyses for lazy languages. Compared to those analyses when applied to the analysis for strict programs, our analysis is more efficient in a sense that our abstract domain has a smaller size.

Among those destructive update analyses for imperative languages, the recent work by Paige and Goyal is a good comparison to our dynamic analysis. They applied their work to the imperative SETL language, which has a special value ω (om) for nullifying variables. The algorithm of our dynamic analysis has the same asymptotic order of complexity as Goyal and Paige's algorithm does. The extension of Goyal and Paige's algorithm to an interprocedural analysis is not clear yet, while our analysis has been successfully applied to the analysis of interprocedural cases.

Sestoft's work of replacing the function parameters by global variables includes some techniques closely related to our analysis, although the goal of his analysis is quite different.

2.8.1 Destructive Update Analysis For Lazy Languages

We discuss three approaches of destructive update analysis for lazy languages: the path analysis by Bloss [Blo89a, Blo89b, Blo94], the evaluation order analysis by Draghicescu and Purushothaman [DP93], and the effect analysis by Odersky [Ode91]. In spite that their major interests have been in lazy languages, these ap-

proaches may be applied to strict languages for a destructive update optimization as well.

Bloss developed the path model for representing order of evaluation of expressions in a lazy sequential functional language so that the order in which expressions are used is shown to be as the order in which they are evaluated [Blo89a, Blo89b, Blo94]. In this model, a destructive update analysis is simple: Compute the set of update paths for each function in a program, and for the occurrence version of each function. Then look at the update paths through the occurrence functions: if in any path in which an update element $(upd_i,\ a)$ occurs there is later another occurrence of a, we conclude that upd_i cannot be done destructively.

Draghicescu and Purushothaman developed a uniform treatment of order of evaluation and aggregate update [DP93]. Under their assumption the expressions evaluate to references each of which is either a reference to a newly created object, or a reference denoted by some variable in the expression. Their destructive update analysis is based on their analysis of the access orders of variables.

Odersky proposed *liabilities* of expressions for checking the side-effect which a function may exert via a destructive update that remains invisible [Ode91]. The effect on an entity x of evaluating an expression e is abstracted to: $\perp$ (not accessed), rd (read), id (potential alias), el (potential element), sh (potential alias

or element), wr (potentially updated), and $\top$ (conflict). The total effect of an expression is abstracted to a liability, mapping from the program's bound variables to pairs of uses $u_1.u_2$, where u_1 represents the effect on e of x and u_2 represents the effect of e on x.

The static destructive update analysis for lazy languages is expensive. The following table shows the summary of the comparison among these approaches on the number of possible sets for the arguments of an n-ary function f in their abstract domains.

analysis	size of powerdomain
Path (Bloss)	$O(2^{N!+(N-1)!+...+1})$
Evaluation order (Draghicescu and Purushothaman)	$O(2^{2^N})$
Effect (Odersky)	$O(2^{2^N})$
Destructive effect	$O(2^N)$

Please note that one of the reasons that our destructive effect analysis has a smaller number of possible sets of the arguments is that our source language has a strict semantics.

2.8.2 Destructive Update Analysis For Imperative Languages

The recent work by Goyal and Paige [GP98] views the variables pointing to the same value as an equivalent class. In such a way, the must-alias problem is treated

as a partitioning problem over the space of variables. They improved the intuitive $O(N^4)$ algorithm for partitioning the set of variables into an $O(N^3)$ algorithm with the work-list technique. They also envisioned a way to abstract variable partitions away from the standard semantics, while some more details than in [GP98] are needed to complete the abstract interpretation.

The goal of their work is to achieve a fast intra-procedural analysis for a destructive update optimization. Therefore, there are no function definition and function application statements in the source language they studied. And, they analyzed the must-alias, instead of may-alias, relationship among variables which result in a more conservative optimization.

When applying their approach to a destructive update optimization, they rely on the dynamic reference counts to decide the liveness of values. They conjectured the static destructive update analysis can be done by the static variable partition analysis together with a static polyvariant analysis for the liveness of values. It still needs more adaption work to see how to achieve a static destructive update optimization by the static variable partition analysis together with the variable partition analysis.

2.8.3 Analysis For Globalization

Sestoft developed an optimization technique of replacing function parameters by global variables [Ses88, Ses89]. The objective of globalization is to reduce the time and space cost of stack (or heap) allocation of function parameters when possible. Although Sestoft did not apply his approach to the analysis for aggregate updates, and the goal of his work is different from ours, we discuss the comparison between our method and Sestoft's approach.

One of the main contributions in Sestoft's work is to introduce the concepts of definition-use (or *du* for a shorthand) path, path semantics, interference, and definition-use grammar. A *definition-use path* is a linear recording of the definitions and uses of variables during a computation. A du-path π has *interference* with respect to variable x if replacing x by a global variable could change the result of the evaluation.

Sestoft assumes copying for each binding situation (and parameter passing) and therefore no sharing between variables. By making a function parameter a global variable, we can benefit when passing a variable which will not be used for the rest of the function. On other hand, our approach may achieve a more aggressive optimization by allowing sharing and allowing destructive updates when safe. We may easily find the examples that if a value is not updated in a function execution, our approach actually makes no copies while Sestoft's approach makes a copy for

passing the parameter each time the function is called.

The way that Sestoft decides whether the globalization is safe is similar to the way that we decide the local destructibility of objects, but we use the time stamps (or program points) in contrast to his use of du paths. The manipulation of du paths for the languages allowing sharing will be more complicated than what Sestoft discussed in his paper.

Please note that Sestoft's analysis is expensive. The complexity of Sestoft's analysis is of the same order of Bloss' path analysis, considering the size of Sestoft's du paths is of the the same order of the size of Bloss' paths.

2.8.4 Analysis For Parallel Languages

We compare our analysis with three well-known approaches for parallel languages: the Sisal project [FCO90, ACC95, ACW96], the way of executing a program on the MIT Tagged-Token Dataflow Architecture, *TTDA*, by Arvind and Nikhil [AN87], and the program transformation for software reusability by Boyle [BM86]. Note that the main goal of these approaches is improving the implementation of parallel programs, instead of eliminating implicit copying.

Sisal [FCO90] was developed by a joint project of the Lawrence Livermore National laboratories and the University of California at Berkeley during the mid 1980's. Sisal has syntax for loops construct on Arrays, which eliminates the pos-

sibility of implicit copying for recursive function calls that implement loops. However, in the Sisal formal semantics [ACC95, ACW96], it is still not clear how to eliminate implicit copying for array updates and for passing arrays as function parameters.

Arvind and Nikhil propose using dataflow graphs as a target for compilation [AN87]. Their source language, *Id*, also has syntax for loops. In order to overcome the deficiencies in purely functional data structures, a special data-structuring mechanism called *I-structure* is introduced into Id. After reducing a program into its dataflow representation, when a dataflow manager responds to a request, it is sent the request to the forwarding address that accompanied the request using the *change_tag* operator. I-structures can be implemented efficiently without complex compile-time or run-time analysis and they do not restrict parallelism. However, write-once structures lack the flexibility of general incrementally updatable structures [Blo89b, ANK87].

Boyle use automated program transformations to derive a Fortran program from a pure functional specification [BCF92, BH91, BM86]. They show that the derived program executed faster on CRAY X-MP than the handwritten Fortran implementation of the same algorithm. They outline 17 major transformational steps for the derivation, and one of the steps is *reducing the complexity of storage usage (implementing reuse of the grip array).* However, they did not specify the

analysis needed for reducing the complexity of storage usage and implementing reuse of the array with preserving the semantics.

2.9 Conclusion

We present destructive effect analysis as a new approach to a destructive update optimization. We include the semantic model of destructive effect analysis for a dynamic destructive update optimization, and the abstract interpretation of destructive effect analysis for a static destructive update optimization. Our analysis uses time stamps for the liveness analysis. The variables in a function sharing the same values in the standard semantics eventually share the same destructive effects in the abstract domain of our analysis.

The algorithm of our dynamic destructive effect analysis has an $O(N^3)$ complexity, but it introduces an $O(N^2)$ overhead for computing a destructive effect at run time. Note that among the conventional approaches to a dynamic destructive update optimization, the best known algorithm was the $O(N^4)$ algorithm by Choi, Burke, and Carini [CBC93]. The recent work by Goyal and Paige proposed an $O(N^3)$ algorithm [GP98] for the intra-procedural analysis.

Compared with the other dynamic optimization techniques, the structure of a program is significant to the complexity of our analysis. For most of the programs in the practical software industry, the depths and the number of branches in

nested if expressions are usually less than 10. For those cases, our analysis actually analyzes programs in time linear to the program length while the other analyses require $O(N^3)$ or more time.

The abstract interpretation of destructive effect analysis has the possible destructive effect sets of size $O(2^N)$ through the analysis for n-ary functions in the abstract domain. It is smaller than the conventional approaches of abstract interpretation frameworks for a destructive update analysis, which have the possible sets of size $O(2^{2^N})$ or more for n-ary functions in their abstract domains. One of the reasons that our static destructive effect analysis applies in a smaller abstract domain is that our source language has a strict semantics while the other techniques may be applied to analyze lazy programs.

Appendix: A Semantic Model of Time Stamping Scheme

Here we give a semantic model of time stamps that labels the expressions of a program with lexicographically ordered time stamps. So that, we may compare two program points in a function by their time stamps θ_1 and θ_2. We denote as $\theta_1 < \theta_2$, and $\theta_1 > \theta_2$, that a single threaded execution goes to θ_1 earlier, or later, than θ_2, respectively; $\theta_1 = \theta_2$ that the two program points happen at the same time; and $\theta_1 \cong \theta_2$ that θ_1 and θ_2 happens exclusively to each other.

We define a time stamp θ as follows.

$$\theta = \langle i,\ \alpha \rangle :: \theta \mid \mathsf{nil}.$$

where i is a sequential numbering of expressions at the current if-level, and α is defined as

$$\alpha = \begin{cases} 1 & \text{if in the consequence of an if expression} \\ 2 & \text{if in the alternative of an if expression} \\ 0 & \text{otherwise.} \end{cases}$$

In the discussion we adopt the following notations. We use $\theta.i(\alpha)$ for a shorthand of $\ldots\langle i,\ \alpha\rangle :: \mathsf{nil}$. We use $\theta(\alpha)$ for a shorthand of $\theta.i(\alpha)$ when the value of i is insignificant. We use $\theta.i$ for a shorthand of $\theta.i(\alpha)$ when $\alpha = 0$.

Semantic Domains

Θ the domain of lexicographically ordered time stamps

Semantic Functions

$$\mathcal{E}_t : [[Exp]] \to \Theta \to \Theta : Exp$$

$$\begin{aligned} \mathcal{E}_t\ [[c]]\ \theta &= \theta : c,\ [Inc(\theta)/\theta] \\ \mathcal{E}_t\ [[x]]\ \theta &= \theta : x,\ [Inc(\theta)/\theta] \\ \mathcal{E}_t\ [[up\ x]]\ \theta &= \theta : up\ x,\ [Inc(\theta)/\theta] \end{aligned}$$

$$
\begin{aligned}
\mathcal{E}_t\ [[np\ x]]\ \theta &= \theta : np\ x,\ [Inc(\theta)/\theta] \\
\mathcal{E}_t\ [[f\ e_1\ \ldots\ e_n]]\ \theta &= f\ (\theta_1 : e_1)\ \ldots\ (\theta_n : e_n),\ [Inc(\theta_n)/\theta] \\
&\quad \text{where } \theta_1 = \theta,\ \text{and } [\theta_{i+1} = Inc(\theta_i)]_1^{n-1} \\
\mathcal{E}_t\ [[\mathsf{let}\ x = e_0\ \mathsf{in}\ e_1]]\ \theta &= \mathsf{let}\ \theta : x = e_0\ \mathsf{in}\ \mathcal{E}_t[[e_1]]\ \theta_1,\ [Inc(\theta_1)/\theta], \\
&\quad \text{where } \theta_1 = Inc(\theta) \\
\mathcal{E}_t\ [[\mathsf{if}\ e_0\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2]]\ \theta &= \mathsf{if}\ \mathcal{E}_t\ [[e_0]]\ \theta\ \mathsf{then}\ \mathcal{E}_t\ [[e_1]]\ \theta(1).1 \\
&\quad \mathsf{else}\ \mathcal{E}_t\ [[e_2]]\ \theta(2).1,\ [Inc(\theta)/\theta]
\end{aligned}
$$

$$
\mathcal{A}_t\ [[\{f_i(x_1, \ldots, x_n) = e_i]]\ \theta = \{f_i(x_1, \ldots, x_n) = (\mathcal{E}_t\ [[e_i]]\ 1)\}
$$

Please note that the time stamps of the expressions in different functions are independent from each other.

We use an auxiliary function $Inc(\theta)$ to compute the time stamp after the execution of an expression.

$$
Inc(\langle i,\ \alpha\rangle :: \theta) = \begin{cases} i + 1 & \text{if } \alpha = 0 \\ \langle i,\ \alpha\rangle :: Inc(\theta) & \text{otherwise.} \end{cases}
$$

The auxiliary function $\mathcal{C}(\theta_1, \theta_2)$ for comparing two time stamps is defined as follows.

$$
\mathcal{C}(\langle i_1,\ \alpha_1\rangle :: \theta_1,\ \langle i_2,\ \alpha_2\rangle :: \theta_2)
$$

$$= \begin{cases} -1 & \text{if } i_1 < i_2 \text{ or } (i_1 = i_2 \text{ and } \alpha_1 = 0 \text{ and } \alpha_2 > 0) \\ 1 & \text{if } i_1 > i_2 \text{ or } (i_1 = i_2 \text{ and } \alpha_2 = 0 \text{ and } \alpha_1 > 0) \\ 0 & \text{if } i_1 = i_2 \text{ and } \alpha_1 = \alpha_2 = 0 \\ 2 & \text{if } i_1 = i_2 \text{ and } \alpha_1 + \alpha_2 = 3 \\ \mathcal{C}(\theta_1,\ \theta_2) & \text{otherwise.} \end{cases}$$

We analyze the complexity of the algorithm of time stamps as follows.

Theorem A. *By scanning a program once, we may give the time stamp to each expression in a program in $O(N \times d)$ time, where d is the maximum depth of nested* if *expressions, and d is $O(N)$ in the worst case.*

Proof: It follows the facts that it takes $O(d)$ time to make a copy of a time stamp, and that there is no recursive evaluation for time stamps. □

Chapter 3

Implementation

In this chapter we present the implementation work of this dissertation, which includes:

- extending the typed λ-calculus with sets,
- designing the syntax and semantics of EAS,
- implementing a dynamic optimizer with destructive effect analysis, and
- implementing a program analyzer with static destructive effect analysis.

"All functional programming languages can be viewed as syntactic variations of the λ-calculus, so that both their semantics and implementation can be analyzed in the context of the λ-calculus [SK95]." In [Yun97], we extended the typed λ-calculus to accommodate the operations with typed sets. Our extension is based

on the typed λ-Calculus described in [NN92].

We designed an experimental applicative language with sets, *EAS*, that implements the SET λ-calculus. EAS (read as ease) is strongly and statically typed. It has a polymorphic type system. This chapter includes a description on the definition of the EAS programming language.

The dynamic optimizer that we implemented works by augmenting the object code with the codes that analyze a program at run time. So that, the program performs an updating expression by a destructive update when the analysis detects that it is safe.

The destructive update analyzer implements the static destructive effect analysis. In contrast with the dynamic optimizer, the destructive effect analyzer performs the analysis without the actual values in the standard domain. This leads to the computation of a set of all possible values for an if expression and the computation of a fixed point for a recursive function in the abstract domain so that the analysis may be performed at compile time.

3.1 Set-Enriched Typed λ-Calculus

This section briefly describes our work on extending the typed λ-calculus to accommodate the operations with typed, or homogeneous, sets. We call the extension as Set-Enriched Typed λ-Calculus, or in short SET λ-calculus. Please refer to

[Yun97] for a more detailed description.

Functions play an essential role in mathematics. Much of the theory of functions, including the issue of computability, unfolds as part of mathematical logic before advent of computers. In particular, Alonzo Church developed the λ-calculus in the 1930s as a theory of functions that provides rules for manipulating functions in a purely syntactic manner [SK95].

Beyond the influence of the λ-calculus in the area of computation theory, it has contributed important results to the formal semantics of programming languages:

- Although the λ-calculus has the power to represent all computable functions, its uncomplicated syntax and semantics provide an excellent vehicle for studying the meaning of programming language concept.

- All functional programming languages can be viewed as syntactic variations of the λ-calculus, so that both their semantics and implementation can be analyzed in the context of the λ-calculus.

- Denotational semantics, one of the foremost methods for a formal specification of programming languages, grew out of the research in the λ-calculus and expresses its definitions using the higher-order functions of the λ-calculus.

Hence, the work of extending the typed λ-calculus to sets is actually the study of feasibility of extending all functional languages with sets.

3.1.1 SET λ-Calculus Syntax

In this section, we present the syntax of the SET λ-calculus. The SET λ-calculus has types, $t \in T$, and expressions, $e \in E$, and its syntax is shown in Figure 3.1. Following the syntax of the typed λ-calculus [NN92], A_i are base types where i ranges over an unspecified index set I and we shall write Bool for the type A_{bool} of booleans; Int for the type A_{int} of integers; and Void for the type A_{void} that intuitively only has a dummy element. Product types, function types, list types, and set types are constructed using the type constructors $\times$, $\rightarrow$, list, and set, respectively. $f_i[t]$ are primitives of type t as indicated. Related to products we has notation for forming pairs and for selecting their components. Associated with function space we have notation for λ-abstraction, application and variables. Related to lists we have notation for constructing lists, for selecting their first components and for testing for emptiness of lists. Associated with sets we have notation for constructing sets, for selecting their set representatives, for testing for emptiness of sets, and for testing for the membership of elements in sets.

We shall consider SET λ-expressions together with their overall types. We use the term *programs* for such specifications. We define the syntactic category $P(E,T)$ of programs as follows:

$$e \in E$$

$$t \in T$$

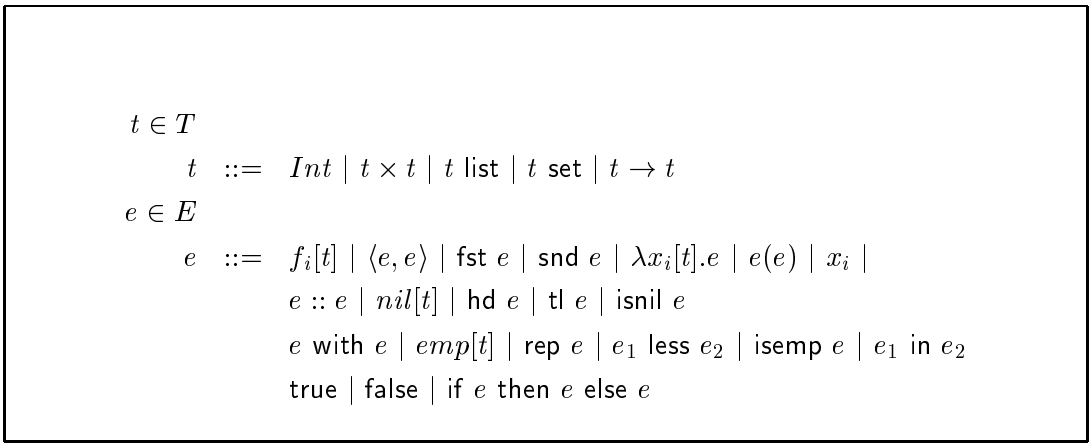


Figure 3.1: The syntax of the Set-Enriched Typed λ-calculus

$$p \quad \in \quad P(E,T)$$

$$p \quad ::= \quad \mathsf{DEF}\ x_i = e\ p \mid \mathsf{VAL}\ e\ \mathsf{HAS}\ t$$

where $P(E,T)$ is the syntactic category of the programs in which all the expressions are SET λ-expressions and their types are as shown above.

When we allow the user to specify a SET λ-expression and its corresponding type by giving an untyped expression, we pretend that we have a function of the functionality $P(UE,T) \to P(E,T)$ that can automatically introduce the required type information into the untyped expressions.

3.1.2 Well-Formedness

We need to be precise about when an expression in the set-enriched typed λ-calculus has a given type. Our goal is to show that our extension preserve the

well-formedness of the typed λ-calculus. The details are given in Figure 3.2. We define a relation $tenv \vdash e : t$ for when the expression e has A type t, where $tenv$ is a *type environment*. We use the notation $tenv[t/x_i]$ for an environment that is like $tenv$ except that it maps the variable x_i to the type t. The well-formedness of a program is then given by

$$\frac{\begin{array}{rcl} \emptyset & \vdash & e_1 : t_1 \\ & \vdots & \\ \emptyset[t_1/x_1]\ldots[t_{n-1}/x_{n-1}] & \vdash & e_n : t_n \\ \emptyset[t_1/x_1]\ldots[t_n/x_n] & \vdash & e : t \end{array}}{\vdash \ \mathsf{DEF}\ x_1 = e_1 \ldots\ \mathsf{DEF}\ x_n = e_n\ \mathsf{VAL}\ e\ \mathsf{HAS}\ t}$$

where we have extended the relation $\vdash$ to work on programs as well as on expressions.

In the syntax of the SET λ-calculus, we have included sufficient type information in an expression for the type of an expression to be determined by the types of its free variables. We may state as follows.

> The relation $tenv \vdash e : t$ is functional in $tenv$ and e; this means that $tenv \vdash e : t_1$ and $tenv \vdash e : t_2$ imply $t_1 = t_2$. $\Box$

Hence, we have shown the following fact:

> An expression in the SET λ-calculus has a given type. $\Box$

In other words, the SET λ-calculus is well-formed.

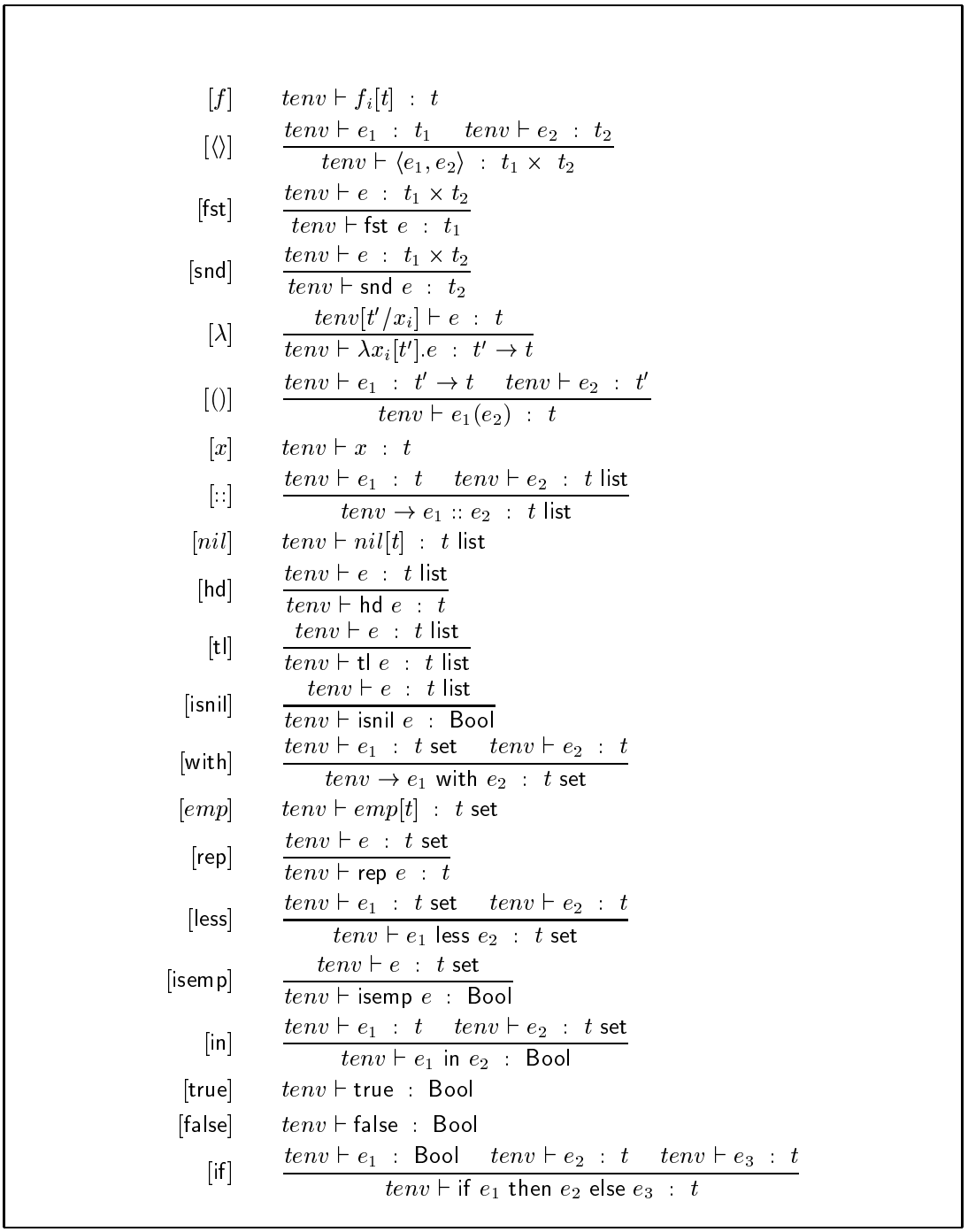


Figure 3.2: Well-formedness of the SET λ-calculus

3.1.3 Type Analysis

In this section, we consider how to propagate type information into an untyped λ-expression so as to obtain an expression in the set-enriched typed λ-calculus. Our goal is to show that our extension preserves the soundness and the completeness of the type analysis for the typed λ-calculus.

We consider the type analysis as a transformation

$$\mathcal{P} : P(UE, T) \to P(E, T).$$

We introduce a syntactic category of *type variables* as follows.

$$\begin{aligned} tv &\in TV \\ tv &::= X_1 \mid X_2 \mid \ldots \end{aligned}$$

A *polytype* is then like a type except that it may contain type variables and this motivates the following definition:

$$\begin{aligned} pt &\in PT \\ pt &::= A_i \mid pt \times pt \mid pt \to pt \mid pt\ \mathsf{list} \mid pt\ \mathsf{set} \mid tv \end{aligned}$$

We write $FTV(pt)$ for the set of free variables in pt. If $FTV(pt) = \emptyset$ we say that pt is a *monotype*. A *type scheme* then is a polytype where some of the type variables may be universally bound. More precisely,

$$ts \in TS$$

$$ts \ ::= \ pt \mid \forall tv.ts$$

A type scheme with an empty set of free type variables is said to be *closed*.

Closely related to polytypes is the notation of a *substitution*. For a total function $S : TV \to PT$ to qualify as a substitution we require that the set $Dom(S) = \{tv \in TV | S(tv) \neq tv\}$ is *finite*. Hence, we could present a substitution S as $[pt_1/X_1, \ldots, pt_n/X_n]$ where $Dom(S) = \{X_1, \ldots, X_n\}$ and $S(X_i) = pt_i$. We will simply write $S(pt)$ for the result of applying the substitution S to the polytype pt. We may thus regard a substitution S as a total function $S : PT \to PT$ and this allows us to define the composition of substitutions as a merely functional composition. So, we say $FTV(S)$, for the free type variables that S uses as

$$FTV(S) = \bigcup \{FTV(S(tv)) \mid tv \in Dom(S)\}.$$

If the set $FTV(S)$ is empty we shall say that S is a *ground substitution*.

An *instance* of a polytype pt for a substitution S is a polytype of the form $S(pt)$. We shall say that S *covers* pt if $Dom(S) \supseteq FTV(pt)$. When S covers pt and S is a ground substitution, the polytype $S(pt)$ will be a monotype because all type variables in pt will be replaced by monotypes. Turning to a closed type scheme $ts = \forall tv_1.\cdots.\forall tv_n.pt$, an instance of ts is simply an instance of pt. A *generic instance* of the closed type scheme ts is an instance pt' of pt such that pt is also an instance of pt'.

Lemma 1. (Robinson) There exists an algorithm U which, when supplied with a set $\{pt_1, \ldots, pt_n\}$ of polytypes, either fails or produces a substitution S:

- It fails if and only if there exists no substitution S such that $S(pt_1) = \ldots = S(pt_n)$.
- If it produces a substitution S then
  - S unifies $\{pt_1, \ldots, pt_n\}$, i.e. $S(pt_1) = \ldots = S(pt_n)$,
  - S is a most general unifier for $\{pt_1, \ldots, pt_n\}$, i.e. whenever $S'(pt_1) = \ldots = S'(pt_n)$ for a substitution S' there exists a substitution S'' such that $S' = S'' \circ S$,
  - S only involves type variables in the pt_i; or more formally expressed as $Dom(S) \cup FTV(S) \subseteq \bigcup_{i=1}^{n} FTV(pt_i)$. $\Box$ [Rob65]

We now present the transformation from untyped programs in $P(UE, T)$ into typed programs in $P(E, T)$. We allow the SET λ-expression e to be polytyped. In other words, rather than producing an expression $e \in E$ we shall transform into a polytyped expression $pe \in PE$ where the syntactic category of polytyped expressions is given by

$$pe \quad \in \quad PE$$

$$pe \quad ::= \quad f_i[pt] \mid \ldots \mid \textsf{if Bool then } pt \textsf{ else } pt.$$

Clearly, if we remove the polytypes in *pe* we must obtain the untyped λ-expression *ue* that we start with.

The functionality of the type analysis algorithm $\mathcal{E}_{ta}$ thus is

$$\mathcal{E}_{ta} : UE \to PE \times pt$$

while it is helpful to obtain also the overall polytype of the polytyped expression. This function is partial, i.e. is allowed to *fail* for it will use the algorithm U in Lemma 1, which is also allowed to fail.

We define an assumption function $A : PE \to P_{fin}(\{x_i | i \in I\} \times PT)$ that extracts the required information. A is a finite subset of $\{x_i | i \in I\}$. We shall write $FTV(A) = \bigcup\{FTV(pt) \mid (x_i, pt) \in A\}$ for the type variables that A uses, $S \circ A = \{(x_i, S(pt) \mid (x_i, pt) \in A\}$ for the result of applying a substitution S to an assumption A, $A_{\overline{x}} = \{(x_i, pt) \in A \mid x_i \neq x\}$ for the part of A that does not involve the variable x, and $A(x) = \{pt \mid (x, pt) \in A\}$ for the set of polytypes that A associates with x. We say that A is *functional* if each $A(x)$ is empty or singleton.

We now define the type analysis algorithm $\mathcal{E}_{ta}$ in Figures 3.3, 3.4, and 3.5 by a structural induction in which we ask for *fresh* variables, i.e. type variables that have not been used before. Also, we extend substitutions to work on polytypes. $\mathcal{E}_{ta}$ inspects its arguments in a bottom-up manner. For each occurrence of a variable it introduces a fresh variable. These type variables may be replaced by other polytypes as a result of unifying the polytypes of various sub-expressions. However,

only when the enclosing λ-abstraction is encountered will the polytypes of the variable occurrences be unified.

We may now define the transformation function $\mathcal{P}$ for programs has the functionality

$$\mathcal{P} : P(UE, T) \to P(E, T).$$

$\mathcal{P}$ is partial because it invokes $\mathcal{E}_{ta}$ and U, and these two algorithms may fail. The definition of $\mathcal{P}$ is

$$\begin{aligned}
& \mathcal{P}[[\mathsf{DEF}\ x_i = ue_1\ \ldots\ \mathsf{DEF}\ x_n = ue_n\ \mathsf{VAL}\ ue_0\ \mathsf{HAS}\ t]] \\
= \ & \mathsf{let}\ ((\lambda x_1[pt_1].\cdots.(\lambda x_n[pt_n].pe_0)(pe_n)\ldots(pe_1)), pt) = \\
& \quad \mathcal{E}[[\lambda x_1.\cdots.(\lambda x_n.ue_0)(ue_n)\ldots(ue_1)]] \\
& \quad \mathsf{let}\ S_1 = U(\{pt, t\}) \\
& \quad \mathsf{let}\ S_2 = (\lambda tv.\mathsf{Void}) \circ S_1 \\
& \quad \mathsf{in}\ \mathsf{DEF}\ x_1 = \mathcal{E}_e(S_2(pe_1))\ldots\mathsf{DEF}\ x_n = \mathcal{E}_e(S_2(pe_n)) \\
& \quad\quad \mathsf{VAL}\ \mathcal{E}_e(S_2(pe_0))\ \mathsf{HAS}\ t
\end{aligned}$$

To obtain the desired result we need to unify the overall polytype produced by $\mathcal{E}_{ta}$ with the monotype t supplied. Also, we need to get rid of any remaining type variables and this motivates the substitutions $\lambda tv.\mathsf{Void}$ that replaces type variables with the uninteresting type Void. Finally, the expression $S_2(pe_i)$ are not yet in E because variables in $S_2(pe_i)$ will be annotated with their types and this is not the

$$
\begin{array}{rcl}
\mathcal{E}_{ta}[[f_i]] & = & \begin{cases} (f_i[pt], pt) & \text{if } C(f_i) = ts \\ & \quad \text{and } pt \text{ is a generic instance of } ts \\ & \quad \text{and } FTV(pt) \text{ are all fresh} \\ fail & \text{otherwise} \end{cases} \\
\mathcal{E}_{ta}[[\langle ue_1, ue_2 \rangle]] & = & \text{let } (pe_1, pt_1) = \mathcal{E}_{ta}[[ue_1]] \\
 & & \text{let } (pe_2, pt_2) = \mathcal{E}_{ta}[[ue_2]] \\
 & & \text{in } (\langle pe_1, pe_2 \rangle, pt_1 \times pt_2) \\
\mathcal{E}_{ta}[[\mathsf{fst}\ ue]] & = & \text{let } (pe, pt) = \mathcal{E}_{ta}[[ue]] \\
 & & \text{let } tv_1 \text{ and } tv_2 \text{ be fresh} \\
 & & \text{let } S = U(\{pt, tv_1 \times tv_2\}) \\
 & & \text{in } (\mathsf{fst}\ S(pe), S(tv_1)) \\
\mathcal{E}_{ta}[[\mathsf{snd}\ ue]] & = & \text{let } (pe, pt) = \mathcal{E}_{ta}[[ue]] \\
 & & \text{let } tv_1 \text{ and } tv_2 \text{ be fresh} \\
 & & \text{let } S = U(\{pt, tv_1 \times tv_2\}) \\
 & & \text{in } (\mathsf{snd}\ S(pe), S(tv_2)) \\
\mathcal{E}_{ta}[[\lambda x_i.ue]] & = & \text{let } (pe, pt) = \mathcal{E}_{ta}[[ue]] \\
 & & \text{let } tv \text{ be fresh} \\
 & & \text{let } S = U(\{tv\} \cup A(pe)(x_i)) \\
 & & \text{in } (\lambda[S(tv)].S(pe), S(tv) \to S(pt)) \\
\mathcal{E}_{ta}[[ue_1(ue_2)]] & = & \text{let } (pe_1, pt_1) = \mathcal{E}_{ta}[[ue_1]] \\
 & & \text{let } (pe_2, pt_2) = \mathcal{E}_{ta}[[ue_2]] \\
 & & \text{let } tv \text{ be fresh} \\
 & & \text{let } S = U(\{pt_1, pt_2 \to tv\}) \\
 & & \text{in } (S(pe_1)(S(pe_2)), S(tv)) \\
\mathcal{E}_{ta}[[x_i]] & = & \text{let } tv \text{ be fresh} \\
 & & \text{in } (x_i[tv], tv)
\end{array}
$$

Figure 3.3: Type analysis of set-enriched typed λ-expressions (part 1)

$$
\begin{array}{rcl}
\mathcal{E}_{ta}[[ue_1 :: ue_2]] & = & \text{let } (pe_1, pt_1) = \mathcal{E}_{ta}[[ue_1]] \\
& & \text{let } (pe_2, pt_2) = \mathcal{E}_{ta}[[ue_2]] \\
& & \text{let } S = U(\{pt_1 \text{ list}, \ pt_2\}) \\
& & \text{in } (S(pe_1) :: S(pe_2), S(pt_2)) \\
\mathcal{E}_{ta}[[nil]] & = & \text{let } tv \text{ be fresh} \\
& & \text{in } (nil[tv], tv \text{ list}) \\
\mathcal{E}_{ta}[[\mathsf{hd}\ ue]] & = & \text{let } (pe, pt) = \mathcal{E}_{ta}[[ue]] \\
& & \text{let } tv \text{ be fresh} \\
& & \text{let } S = U(\{pt, \ tv \text{ list}\}) \\
& & \text{in } (\mathsf{hd}\ S(pe), S(tv)) \\
\mathcal{E}_{ta}[[\mathsf{tl}\ ue]] & = & \text{let } (pe, pt) = \mathcal{E}_{ta}[[ue]] \\
& & \text{let } tv \text{ be fresh} \\
& & \text{let } S = U(\{pt, \ tv \text{ list}\}) \\
& & \text{in } (\mathsf{tl}\ S(pe), S(pt)) \\
\mathcal{E}_{ta}[[\mathsf{isnil}\ ue]] & = & \text{let } (pe, pt) = \mathcal{E}_{ta}[[ue]] \\
& & \text{let } tv \text{ be fresh} \\
& & \text{let } S = U(\{pt, \ tv \text{ list}\}) \\
& & \text{in } (\mathsf{isnil}\ S(pe), \mathsf{Bool}) \\
\mathcal{E}_{ta}[[ue_1 \text{ with } ue_2]] & = & \text{let } (pe_1, pt_1) = \mathcal{E}_{ta}[[ue_1]] \\
& & \text{let } (pe_2, pt_2) = \mathcal{E}_{ta}[[ue_2]] \\
& & \text{let } S = U(\{pt_1, \ pt_2 \text{ set}\}) \\
& & \text{in } (S(pe_1) \text{ with } S(pe_2), S(pt_1)) \\
\mathcal{E}_{ta}[[emp]] & = & \text{let } tv \text{ be fresh} \\
& & \text{in } (emp[tv], tv \text{ set}) \\
\mathcal{E}_{ta}[[\mathsf{rep}\ ue]] & = & \text{let } (pe, pt) = \mathcal{E}_{ta}[[ue]] \\
& & \text{let } tv \text{ be fresh} \\
& & \text{let } S = U(\{pt, \ tv \text{ set}\}) \\
& & \text{in } (\mathsf{rep}\ S(pe), S(tv))
\end{array}
$$

Figure 3.4: Type analysis of set-enriched typed λ-expressions (part 2)

$$
\begin{array}{rcl}
\mathcal{E}_{ta}[[ue_1 \text{ less } ue_2]] & = & \text{let } (pe_1, pt_1) = \mathcal{E}_{ta}[[ue_1]] \\
& & \text{let } (pe_2, pt_2) = \mathcal{E}_{ta}[[ue_2]] \\
& & \text{let } S = U(\{pt_1,\ pt_2 \text{ set}\}) \\
& & \text{in } (S(pe_1) \text{ less } S(pe_2), S(pt_1)) \\
\mathcal{E}_{ta}[[\text{isemp } ue]] & = & \text{let } (pe, pt) = \mathcal{E}_{ta}[[ue]] \\
& & \text{let } tv \text{ be fresh} \\
& & \text{let } S = U(\{pt,\ tv \text{ set}\}) \\
& & \text{in } (\text{isemp } S(pe), \text{Bool}) \\
\mathcal{E}_{ta}[[ue_1 \text{ in } ue_2]] & = & \text{let } (pe_1, pt_1) = \mathcal{E}_{ta}[[ue_1]] \\
& & \text{let } (pe_2, pt_2) = \mathcal{E}_{ta}[[ue_2]] \\
& & \text{let } S = U(\{pt_1 \text{ set},\ pt_2\}) \\
& & \text{in } (S(pe_1) \text{ in } S(pe_2), \text{Bool}) \\
\mathcal{E}_{ta}[[\text{true}]] & = & (\text{true},\ \text{Bool}) \\
\mathcal{E}_{ta}[[\text{false}]] & = & (\text{false},\ \text{Bool}) \\
\mathcal{E}_{ta}[[\text{if } ue_1 \text{ then } ue_2 \text{ else } ue_3]] & = & \text{let } (pe_1, pt_1) = \mathcal{E}_{ta}[[ue_1]] \\
& & \text{let } (pe_2, pt_2) = \mathcal{E}_{ta}[[ue_2]] \\
& & \text{let } (pe_3, pt_3) = \mathcal{E}_{ta}[[ue_3]] \\
& & \text{let } S_1 = U(\{pt_1,\ \text{Bool}\}) \\
& & \text{let } S_2 = U(\{pt_2,\ pt_3\}) \\
& & \text{in } (\text{if } S_1(pe_1) \text{ then } S_2(pe_2) \\
& & \quad \text{else } S_2(pe_3), S_2(pt_2))
\end{array}
$$

Figure 3.5: Type analysis of set-enriched typed λ-expressions (part 3)

case for the expression in E. We therefore need to use the type erasing function $\mathcal{E}_e$ that has a functionality of $\mathcal{E}_e(x_i[t]) = x_i$ for the variables with type information, but otherwise behaves as the identity.

3.1.4 Soundness and Completeness

In this section, we discuss the soundness and the completeness of the type analysis function $\mathcal{E}_{ta}$. We need two auxiliary sets, $WFF(ue)$ and $INS(ue)$. The set

$$WFF(ue) = \{\langle tenv, e, t\rangle \mid tenv \vdash e : t \wedge ue = \mathcal{E}_e(e) \wedge Dom(tenv) = FEV(ue)\}$$

is a set of triples of the form $\langle tenv, e, t\rangle$ such that e has a type t; that is, $tenv \vdash e : t$, and e equals ue when the types are erased by the type erasing function $\mathcal{E}_e : PE \rightarrow UE$, and where $FEV(ue)$ is the set of free variables in ue. Analogously, the set

$$\begin{aligned} INS(ue) \;=\; & \{\langle S \circ A(pe), \mathcal{E}_e(S(pe)), S(pt)\rangle \\ & \mid \mathcal{E}_{ta}[[ue]] = (pe, pt) \wedge Dom(S) \supseteq FTV(pe) \cup FTV(pt) \\ & \wedge\ S \text{ is a ground substitution} \wedge S \circ A(pe) \text{ is functional}\} \end{aligned}$$

is a set of triples of the form $\langle S \circ A(pe), \mathcal{E}_e(S(pe)), S(pt)\rangle$, when $\mathcal{E}[[ue]] = (pe, pt)$. The type is obtained as $S(pt)$, that is a ground substitution applied to polytype pt. The expression would similarly be $S(pe)$, except that polytyped variables are annotated with their types so that we have to use the type erasing function $\mathcal{E}_e$. Finally, the analogue of the type environment is $S \circ A(pe)$ where it is required that

this set may be regarded as a function. Please note that $INS(ue)$ is the empty set, $\emptyset$, if $\mathcal{E}_{ta}[[ue]]$ fails.

We now show a conjunction of two results. $WFF(ue) \supseteq INS(ue)$ is the soundness, which says that $\mathcal{E}_{ta}$ only specifies will-formed expressions; the other result is the completeness, $WFF(ue) \subseteq INS(ue)$, which says that every well-formed expression is obtainable from the result that $\mathcal{E}_{ta}$ specifies.

Theorem 2. (Soundness and Completeness of $\mathcal{E}_{ta}$)

$$WFF(ue) = INS(ue)$$

for all expressions $ue \in UE$.

Note that the correctness of program translating follows, with the introduction of type erasing function $\mathcal{P}_e : P(E,T) \rightarrow P(UE,T)$, defined by

$$\begin{aligned} & \mathcal{P}_e[[\mathsf{DEF}\ x_1 = e_1 \ldots \mathsf{DEF}\ x_n = e_n\ \mathsf{VAL}\ e\ \mathsf{HAS}\ t]] \\ = \quad & \mathsf{DEF}\ x_i = \mathcal{E}_e[[e_1]] \ldots \mathsf{DEF}\ x_n = \mathcal{E}_e[[e_n]]\ \mathsf{VAL}\ \mathcal{E}_e[[e]]\ \mathsf{HAS}\ t \end{aligned}$$

Proof. We prove by a structural induction on ue using the *freshness* of the type variables generated in $\mathcal{E}_{ta}$.

- The case $ue ::= f_i$. If $f_i \notin Dom(C)$, both $WFF(ue)$ and $INS(ue)$ are empty, where C is the set of constraints. Assuming $f_i \in Dom(C)$, we have

  $$WFF(ue) \quad = \quad \{\langle \emptyset, f_i[t], t\rangle \mid t \text{ is an instance of } C(f_i)\}$$

$$INS(ue) = \{\langle \emptyset, f_i[S(pt)], S(pt)\rangle \mid S \text{ covers } pt,\ S \text{ is ground},\ pt \text{ is a generic instance of } C(f_i) \text{ with } FTV(pt) \text{ all being fresh}\}$$

where $C(f_i)$ is the set of all constraints on f_i. The result then follows because the set of types that are instances of a closed type scheme equals the set of types that are ground instances of a polytype that is a generic instance of the same closed types scheme.

- The case $ue ::= \lambda x_i.ue_0$. We assume that $x_i \in FEV(ue_0)$. We then have

$$WFF(ue) = \{\langle tenv, \lambda x_i[t].e_0, t \to t_0\rangle \mid \langle tenv[t/x_i], e_0, t_0\rangle \in WFF(ue_0) \ \wedge Dom(tenv) = FEV(ue)\}$$

because the instance system in Figure 3.2 is such that $[\lambda]$ must be the last rule used in a proof of well-formedness of $\lambda x_i.e_0$. We write $tenv/X$ for the restriction of an environment $tenv$ to a subset X of its domain $Dom(tenv)$. We have

$$WFF(ue) = \{\langle tenv/FEV(ue),\ \lambda x_i[tenv(x_i)].e_0, tenv(x_i) \to t_0\rangle \mid \langle tenv, e_0, t_0\rangle \in WFF(ue_0)\}.$$

For $INS(ue)$ we have

$$INS(ue) = \{\langle (S \circ S') \circ A(pe_0)_{x_i},\ \lambda x_i[(S \circ S')(tv)].\varepsilon'((S \circ S')(pe_0)),$$

$$
\begin{aligned}
& (S \circ S')(tv) \rightarrow (S \circ S')(pt_0)\rangle \\
\mid \quad & \mathcal{E}[[ue_0]] = (pe_0, pt_0) \wedge tv \text{ is fresh } \wedge\ U\{tv\} \cup A(pe_0)_{x_i} = S' \\
& \wedge\ S \text{ covers all of } S' \circ A(pe_0)_{x_i},\ \ S'(tv),\ S'(pe_0),\ \textit{and}\ S'(tv) \\
& \wedge\ S \text{ is ground } \wedge\ (S \circ S') \circ A(pe_0)_{x_i} \text{ is functional } \}
\end{aligned}
$$

where, we have used

$$
\begin{aligned}
(S \circ S') \circ A(pe_0)_{x_i} &= S \circ (S' \circ A(pe_0)_{x_i}), \\
(S \circ S')(pe_0) &= S(S'(pe_0)), \\
(S \circ S')(pt_0) &= S(S'(pt_0)), \text{ and} \\
A(\lambda x_i[tv].pe_0) &= A(pe_0)_{x_i}.
\end{aligned}
$$

Since S' unifies the set $A(pe_0)(x_i)$, the set $((S \circ S') \circ A(pe_0))(x_i)$ is a singleton. Hence, $(S \circ S')(tv)$ may be replaced by $((S \circ S') \circ A(pe_0))(x_i)$. And, since $FEV(ue) = FEV(ue_0) \backslash \{x_i\}$ where $\backslash$ is the set difference, we may replace $(S \circ S') \circ A(pe_0)_{x_i}$ by $((S \circ S') \circ A(pe_0))/FEV(ue)$. Thus, we have

$$
\begin{aligned}
INS(ue) = \ & \{(((S \circ S') \circ A(pe_0))/FEV(ue), \\
& \lambda x_i[((S \circ S') \circ A(pe))(x_i)].\ \varepsilon'((S \circ S')(pe_0)), \\
& ((S \circ S') \circ A(pe_0))(x_i) \rightarrow (S \circ S')(pt_0)) \\
\mid \quad & \mathcal{E}[[ue_0]] = (pe_0, pt_0)\ \wedge ((S \circ S') \circ A(pe_0))(x_i) \text{ is a singleton} \\
& \wedge (S \circ S') \text{ covers all of } A(pe_0)_{x_i}, A(pe_0)(x_i), pe_0 \text{ and } pt_0
\end{aligned}
$$

$\wedge(S \circ S')$ is ground

$\wedge((S \circ S') \circ A(pe_0))/FEV(ue)$ is functional$\}$

where the need for tv vanished, given that $((S \circ S') \circ A(pe_0))(x_i)$ has replaced $(S \circ S')(tv)$. We then get

$$\begin{aligned} INS(ue) &= \{(tenv/FEV(ue),\ \lambda x_i[tenv(x_i)].e_0,\ tenv(x_i) \to t_0) \\ &\mid (tenv, e_0, t_0) \in INS(ue_0)\}. \end{aligned}$$

So, the desired result follows from the induction hypothesis. Finally, we observe that if $x_i \notin FEV(ue_0)$ the result may be obtained in much the same manner.

- We observe that the desired results for the following cases follow the pattern of the above cases:

  $ue ::= \langle ue_1, ue_2 \rangle$, $ue ::= \mathsf{fst}\ ue_0$, $ue ::= \mathsf{snd}\ ue_0$,

  $ue ::= ue_1(ue_2)$, $ue ::= x_i$, $ue ::= ue_1 :: ue_2$,

  $ue ::= nil$, $ue ::= \mathsf{hd}\ ue_0$, $ue ::= \mathsf{tl}\ ue_0$,

  $ue ::= \mathsf{isnil}\ ue_0$, and $\mathsf{if}\ ue_1\ \mathsf{then}\ ue_2\ \mathsf{else}\ ue_3$.

- The cases

  $ue ::= e_1\ \mathsf{with}\ e_2$, $ue ::= isemp$, $ue ::= \mathsf{rep}\ ue_0$,

  $ue ::= ue_1\ \mathsf{less}\ ue_2$, $ue ::= \mathsf{isemp}\ ue_0$

follow, respectively, the general patterns of the cases

$$ue ::= ue_1 :: ue_2, \quad ue ::= nil, \qquad ue ::= \mathsf{hd}\ ue_0,$$

$$ue ::= \mathsf{tl}\ ue_0, \qquad ue ::= \mathsf{isnil}\ ue_0.$$

- Finally, the case of $ue ::= ue_1\ \mathsf{in}\ ue_2$ follows the general pattern of $ue ::= ue_1\ \mathsf{with}\ ue_2$ with replacing $t\ \mathsf{set}$ by Bool. □

Theorem 3. (Soundness and Completeness of P) Consider an untyped program $up = \mathsf{DEF}\ x_1 = ue_1 \ldots \mathsf{DEF}\ x_n = ue_n\ \mathsf{VAL}\ ue\ \mathsf{HAS}\ t]]$, and consider the outcome of $P[[up]]$. If it produces the program $p = \mathsf{DEF}\ x_1 = e_1 \ldots \mathsf{DEF}\ x_n = e_n\ \mathsf{VAL}\ e\ \mathsf{HAS}\ t$ then

- p is well-formed, i.e. $\vdash p$,
- the underlying program of p is up, i.e. $up = \mathcal{P}_e(p)$.

If $p[[up]]$ fails then

- there is no well-formed program p in $P(E, T)$ that has up as its underlying program.

Proof. This is a straightforward corollary of Theorem 2. □

3.2 EAS

As for the study of programming languages, the language definitions usually consist of two major components [SK95]:

1. **Syntax** refers to the ways symbols may be combined to create well-formed sentences or programs in the language. Syntax defines the formal relations between the constituents of a language, thereby providing a structural description of the various expressions that make up legal strings in the language.

2. **Semantics** reveals the meaning of syntactically valid strings in a language. For a programming language, semantics describes the behavior that a computer follows when executing a program in the language.

This section is a brief description on both the syntax and the semantics of EAS, an experimental programming language with sets.

Every programming language presents its own conceptual view of computations. We note the highlights of the EAS programming language as follows:

- EAS is an *applicative* programming language. The principal control mechanism in EAS is recursive function applications.

- EAS is *strongly typed*. Every valid EAS expression has a type which is determined automatically by the type system. Strong typing guarantees that no program can incur a type error at run time.

- EAS has a *polymorphic* type system. Each valid EAS phrase has a uniquely determined and most general typing that determines the set of contexts in which that the EAS phrase may be legally used.

- EAS is *statically scoped*. EAS resolves identifier references at compile time, leading to more modular and more efficient object programs.

- EAS includes syntax for list constructors and set constructors.

An EAS program consists of a sequence of declarations; the execution of each declaration modifies the environment. In the execution of a declaration there are three phases:

1. *Parsing* determines the grammatical form of a declaration.

2. *Elaboration* is the static phase which determines whether it is well-typed and well-formed.

3. *Evaluation* is the dynamic phase which determines the value of the declaration in the environment.

Usually, we call the combination of the first two phases as *compilation*. Once a declaration is compiled we may evaluate it repeatedly with re-evaluation.

In the following discussion, we adapt the notation used in [MTH90]. In following section, we present the syntax of the EAS programming language. We describe, in section 3.2.2, the EAS static semantics. Section 3.2.3 describes the EAS dynamic semantics.

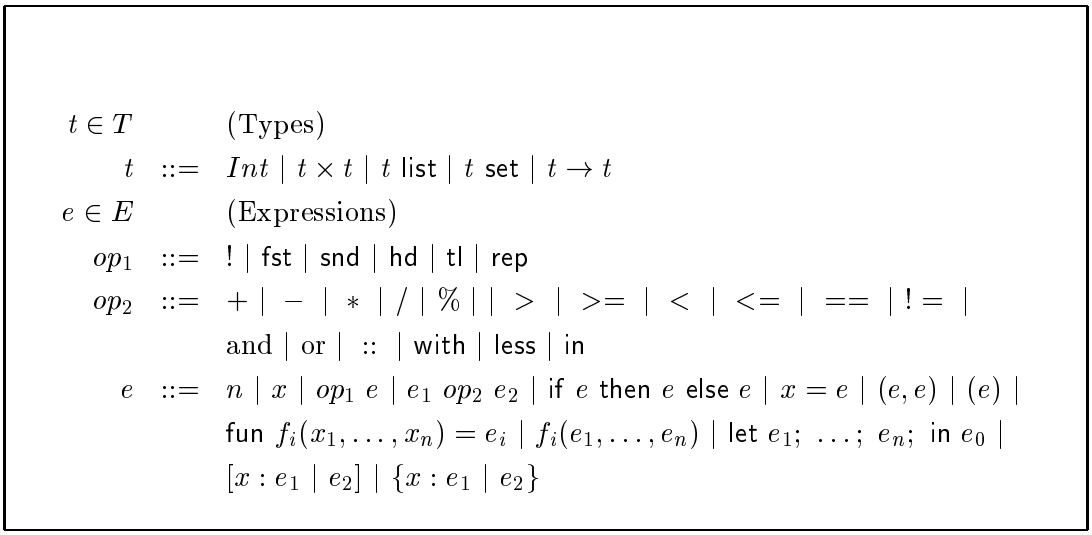


Figure 3.6: The EAS syntax

3.2.1 The EAS Syntax

In this section, we present the EAS syntax. The EAS syntax is a modified version of the SET λ-calculus for the implementation practice purpose. Here are a couple of highlighted modifications:

- EAS adopts the C convention of simulating boolean values with integers.
- EAS includes literals of derived types.

The EAS syntax is shown in Figure 3.6.

Reserved Words

The following are *reserved words* used in EAS. They may not be used as identifiers.

and	else	fun	fst	if	hd
in	less	let	or	rep	snd
tl	with	!	( )	[ ]	{ }
,	::	;	=	==	>
>=	<	>=	+	−	*
/	!=	++	**	|	%

Literals

EAS has one base data type, *integer*. Hence, EAS has one type of base literals, *integer literal*. An integer literal is any non-empty sequence of digits, possibly preceded by a negation symbol −. Examples: 23, −45.

EAS has two kinds of derived lexical forms: one for list, and the other for set. They are presented in the following table.

Derived Form	Equivalent Form
$[lit_1, \ldots, lit_n]$	lit_1 :: ... :: lit_n :: nil
$\{lit_1, \ldots, lit_n\}$	emp with lit_1 ... with lit_n

Identifiers

The rule of an EAS identifier *vid* is described as follows:

> A letter followed by any sequence of letters and digits.

The class of EAS identifiers is denoted as Vid.

Lexical Analysis

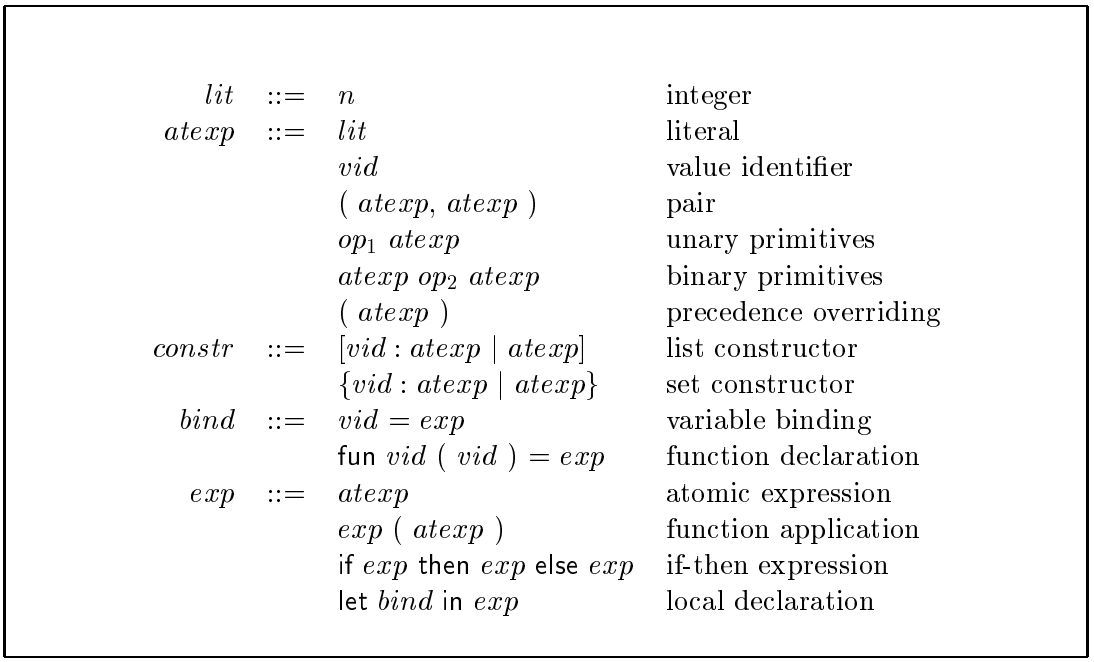
By means of the above rules a compiler can determine the class to which each identifier occurrence belongs. Each item of lexical analysis is either a reserved word, an identifier, or a literal. Comments and formatting characters (such as space, tab, newline, and formfeed) are ignored.

Grammar

EAS has the following phrase classes.

Lit	literals
$AtExp$	atomic expressions
$Constr$	constructors
$Bind$	bindings
Exp	expressions

The grammatical rules for EAS are shown in Figure 3.7. In the EAS grammar, op_1 and op_2 follow the definition presented in the EAS syntax. An EAS program is a sequence of EAS expressions.



lit	::=	n	integer
$atexp$	::=	lit	literal
		vid	value identifier
		($atexp$, $atexp$)	pair
		op_1 $atexp$	unary primitives
		$atexp$ op_2 $atexp$	binary primitives
		($atexp$)	precedence overriding
$constr$	::=	$[vid : atexp \mid atexp]$	list constructor
		$\{vid : atexp \mid atexp\}$	set constructor
$bind$	::=	$vid = exp$	variable binding
		fun vid (vid) = exp	function declaration
exp	::=	$atexp$	atomic expression
		exp ($atexp$)	function application
		if exp then exp else exp	if-then expression
		let $bind$ in exp	local declaration

Figure 3.7: The EAS grammar

3.2.2 The EAS Static Semantics

In addition to the classes of *syntactic objects*, defined in the previous section, we now define the classes of *semantic objects*. The static semantic objects are presented in this section, and the dynamic objects are presented in the following section.

The *static* semantic objects are the object classes manipulated in the elaboration phase of compiling. Some classes contain *simple* semantic objects; such objects are usually identifiers or names of some kind. Other classes contain *compound* semantic objects, such as types or environments, which are constructed from compound objects.

Simple Objects

All semantic objects in the static semantics of the EAS programming language are built from identifiers. EAS has one kind of simple objects: *type constructor names*. The simple semantic object classes and the variables ranging over them are shown as follows.

$$\alpha \text{ or } tyvar \quad \in \quad \mathsf{TyVar} \quad \text{type variables}$$

EAS has only one base type, and therefore $\text{type}(lit) = \mathsf{int}$.

Since EAS does not have user-defined type variables, all the type variables are generated by the compiler during the type analysis. A type variable *tyvar* may be

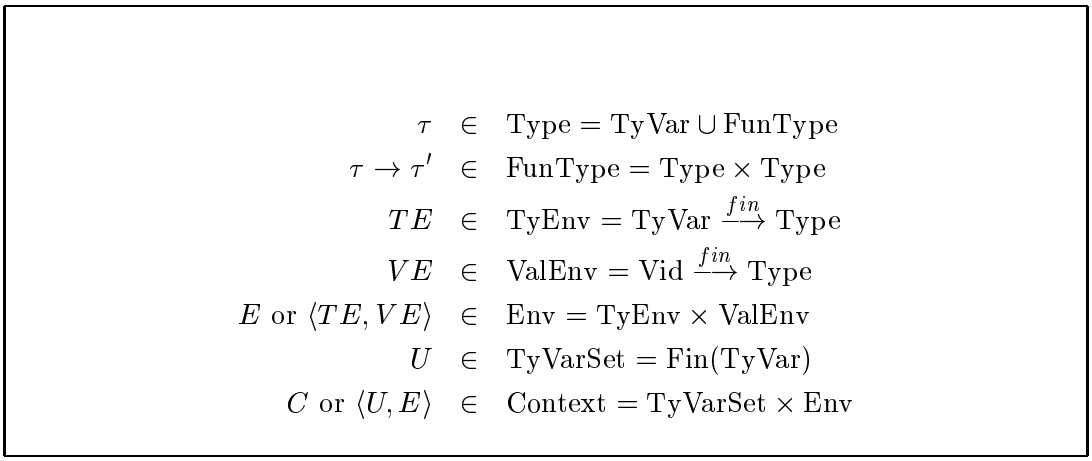


Figure 3.8: EAS Compound Static Semantic Objects

any alphanumeric identifier starting with a prime (').

Compound Objects

The compound objects for the static semantics of EAS are shown in Figure 3.8. We take $\cup$ to mean disjoint union over semantic object classes. We also understand all the defined object classes to be disjoint. We use the notations $A \xrightarrow{fin} B$ for the set of finite maps from A to B, and Fin A for the set of finite subsets of A. The domain and range of a finite map f are denoted as Dom f and Ran f. A finite map is often written explicitly in the form $\{a_1 \mapsto b_1, \ldots, a_k \mapsto b_k\}$, $k \geq 0$; in particular the empty map is $\{\}$.

A value environment VE maps to variable identifiers to types. Note that we treat primitives in the same way as variable identifiers. The types of primitives

Primitive	Type
!	int $\rightarrow$ int
fst	$t_1 \times t_2 \rightarrow t_1$
snd	$t_1 \times t_2 \rightarrow t_2$
hd	t list $\rightarrow t$
tl	t list $\rightarrow t$ list
rep	t set $\rightarrow t$
+, −, ×, /, %, >, >=, <, <=, ==, ! =, and, or	int $\times$ int $\rightarrow$ int
::	$t \times t$ list $\rightarrow t$ list
with, less	t set $\times t \rightarrow t$ set
in	$t \times t$ set $\rightarrow$ int

Figure 3.9: Types of the EAS primitives

are shown in Figure 3.9.

Types and Type Environment

A type τ is one of the following forms:

- int,
- α, and
- $\{lab_1 \mapsto \tau_1, \ldots, lab_n \mapsto \tau_n\}$.

A type environment TE is a finite map from bound type variables tyvar to types t.

Inference Rules

Each rule of the semantics allows inferences among sentences of the form

$$A \vdash phrase \Rightarrow A',$$

where A is usually an environment or a context, *phrase* is an EAS phrase, and A' is a semantic object, usually a type or an environment. It may be pronounced "*phrase* elaborates to A' in context A". Some rules have extra hypotheses not of this form; they are called *side conditions*.

In the presentation of the rules, phrases within single angle brackets $\langle\ \rangle$ are called *options*. To reduce the number of rules, we have adopted the following convention:

> *In each instance of a rule, the options must be either all present or all absent.* □

Atomic Expressions $C \vdash atexp \Rightarrow \tau$

The inference rules of EAS atomic expressions are given as follows.

$$\frac{}{C \vdash lit \Rightarrow \mathrm{type}(lit)} \tag{3.1}$$

$$\frac{C(vid) \succ \tau}{C \vdash vid \Rightarrow \tau} \tag{3.2}$$

$$\frac{C \vdash atexp_1 \Rightarrow \tau_1 \qquad C \vdash atexp_2 \Rightarrow \tau_2}{C \vdash (atexp_1, atexp_2) \Rightarrow \tau_1 \times \tau_2} \tag{3.3}$$

$$\frac{C \vdash atexp \Rightarrow \tau' \quad C(op_1) \succ \tau \to \tau}{C \vdash op_1\ atexp \Rightarrow \tau} \tag{3.4}$$

$$\frac{C(op_2) \succ \tau_1 \times \tau_2 \to \tau \quad C \vdash atexp_1 \Rightarrow \tau_1 \quad C \vdash atexp_2 \Rightarrow \tau_2}{C \vdash atexp_1\ op_2\ atexp_2 \Rightarrow \tau} \tag{3.5}$$

$$\frac{C \vdash atexp \Rightarrow \tau}{C \vdash (atexp) \Rightarrow \tau} \tag{3.6}$$

We note the following:

- In (3.2), the instantiation of type schemes allows different occurrences of a single *vid* to assume different types.
- In (3.3) and (3.4), the type schemes of primitives are predefined in VE.

Constructors $C \vdash constr \Rightarrow \tau$

We present the inference rules of EAS constructors as follows.

$$\frac{C(vid) \succ \tau \quad C \vdash atexp_1 \Rightarrow \tau\ \mathsf{list} \quad C \vdash atexp_2 \Rightarrow \mathsf{int}}{C \vdash [vid : atexp_1 \mid atexp_2] \Rightarrow \tau\ \mathsf{list}} \tag{3.7}$$

$$\frac{C(vid) \succ \tau \quad C \vdash atexp_1 \Rightarrow \tau\ \mathsf{set} \quad C \vdash atexp_2 \Rightarrow \mathsf{int}}{C \vdash \{vid : atexp_1 \mid atexp_2\} \Rightarrow \tau\ \mathsf{set}} \tag{3.8}$$

Value Binding $C \vdash bind \Rightarrow VE$

The inference rules of EAS value bindings are given as follows.

$$\frac{}{C \vdash \quad \Rightarrow \{\} \text{ in Env}} \tag{3.9}$$

$$\frac{C \vdash vid \Rightarrow \langle VE, \tau \rangle \quad C \vdash exp \Rightarrow \tau \quad C \vdash bind \Rightarrow VE'}{C + VE \vdash vid = exp \Rightarrow VE'} \tag{3.10}$$

$$\frac{C \vdash vid_1 \Rightarrow \langle VE, \tau_1 \to \tau_2 \rangle \quad C \vdash vid_2 \Rightarrow \langle VE, \tau_1 \rangle \quad C \vdash exp \Rightarrow \tau_2 \quad C \vdash bind \Rightarrow VE'}{C + VE \vdash \mathsf{fun}\ vid_1(vid_2) = exp \Rightarrow VE'} \tag{3.11}$$

We note that

- In (3.10), when the option is present we have $\mathrm{Dom}(VE) \cap \mathrm{Dom}(VE') = \emptyset$ by the syntactic restrictions.

Expressions $C \vdash exp \Rightarrow \tau$

The inference rules of EAS expressions are given as follows.

$$\frac{C \vdash atexp \Rightarrow \tau}{C \vdash atexp \Rightarrow \tau} \tag{3.12}$$

$$\frac{C \vdash exp \Rightarrow \tau' \to \tau \quad C \vdash atexp \Rightarrow \tau'}{C \vdash exp\ (\ atexp\ ) \Rightarrow \tau} \tag{3.13}$$

$$\frac{C \vdash exp_1 \Rightarrow \mathsf{int} \quad C \vdash exp_2 \Rightarrow \tau \quad C \vdash exp_3 \Rightarrow \tau}{C \vdash \mathsf{if}\ exp_1\ \mathsf{then}\ exp_2\ \mathsf{else}\ exp_3 \Rightarrow \tau} \tag{3.14}$$

$$\frac{C \vdash bind \Rightarrow E \quad C \oplus E \vdash exp \Rightarrow \tau}{C \vdash \mathsf{let}\ bind\ \mathsf{in}\ exp\ \mathsf{end} \Rightarrow \tau} \tag{3.15}$$

Here are some comments:

- In (3.12), the relational symbol $\vdash$ is overloaded for all syntactic classes.
- In (3.15), the use of $\oplus$ ensures that type names generated by the first sub-phrase are different from type names generated by the second sub-phrase.

3.2.3 The EAS Dynamic Semantics

In addition to the classes of *syntactic objects* and *static semantic objects*, we define the classes of *dynamic semantic objects* in this section.

Since types are fully dealt with in the static semantics, the dynamic semantics ignores type information. The EAS syntax is therefore reduced for the purpose of the dynamic semantics. The reduced representation is transformed by removing the asserted type information with the syntax.

Simple Objects

All objects in the dynamic semantics are built from identifier classes together with the simple object classes show as follows.

b	$\in$	BasVal	basic values
op_1	$\in$	Op1Val	unary primitive values
op_2	$\in$	Op2Val	binary primitive values
$constr$	$\in$	ConVal	constructor values
sv	$\in$	SVal	special values

where BasVal is such that each integer denotes a value according to normal mathematical conventions, Op1Val is the class of unary primitive values, Op2Val is the class of binary primitive values, ConVal is the class of constructor values, and SVal is the class of special values. In the EAS dynamic semantics, SVal has only one

element Fail, which is the result of a failing attempt.

Compound Objects

The compound objects for the EAS dynamic semantics EAS are shown as follows.

$$\begin{aligned} v &\in \text{Val} = \text{BasVal} \cup \text{Op1Val} \cup \text{Op2Val} \cup \text{ConVal} \cup \text{SVal} \\ E &\in \text{ValEnv} = \text{Vid} \xrightarrow{fin} \text{Val} \end{aligned}$$

In the presentation of the EAS dynamic compound object, many conventions and notations are adopted in the same way as in the EAS static semantics. We take $\cup$ to mean disjoint union over semantic object classes. We also understand all the defined object classes to be disjoint.

Although with the same names, E, for an environment are used as in the static semantics, the objects denoted are different. We understand this causes no confusion since the static and dynamic semantics are present separately.

Values

The basic values in BasVal are the values bound to predefined variables. The meaning of basic values is presented as the following function:

$$\mathsf{APPLY}_{\mathsf{bas}} : \text{BasVal} \times \text{Val} \to \text{Val} \cup \{\mathsf{Fail}\}.$$

In EAS, we treat primitives and constructors in the same way as basic values. That is, we present the meaning of unary primitive values, binary primitive values,

and constructor values by the following functions:

$$\mathsf{APPLY}_{\mathsf{op}_1} : \mathrm{Op1Val} \times \mathrm{Val} \to \mathrm{Val} \cup \{\mathsf{Fail}\},$$

$$\mathsf{APPLY}_{\mathsf{op}_2} : \mathrm{Op2Val} \times \mathrm{Val} \times \mathrm{Val} \to \mathrm{Val} \cup \{\mathsf{Fail}\}, \text{ and}$$

$$\mathsf{APPLY}_{\mathsf{constr}} : \mathrm{ConVal} \times \mathrm{Vid}\ \mathit{times} \mathrm{Val} \times \mathrm{Val} \to \mathrm{Val} \cup \{\mathsf{Fail}\}.$$

Inference Rules

The dynamic semantic rules allow sentences of the form

$$A \vdash \mathit{phrase} \Rightarrow A'$$

to be inferred, where A is usually an environment, and A' is some semantic object. Some hypotheses in the rules are not of this form, and they are called *side conditions*. The convention for options follows from that of the EAS static semantics.

Another convention adopted in our discussion is that we allow compound variables to range over unions of semantic objects. For instance, we allow x/Fail to range over $X \cup \{\mathsf{Fail}\}$ where x ranges over X.

Atomic Expressions $C \vdash \mathit{atexp} \Rightarrow v/\mathsf{Fail}$

The dynamic semantic inference rules of EAS atomic expressions are given as follows.

$$\frac{}{E \vdash \mathit{lit} \Rightarrow \mathrm{value}(\mathit{lit})} \tag{3.16}$$

$$\frac{E(vid) = v}{E \vdash vid \Rightarrow v} \tag{3.17}$$

$$\frac{E \vdash atexp \Rightarrow v' \qquad \mathsf{APPLY}_{\mathsf{op}_1}(op_1, v') = v}{E \vdash op_1\ atexp \Rightarrow v} \tag{3.18}$$

$$\frac{E \vdash atexp_1 \Rightarrow v_1 \qquad E \vdash atexp_2 \Rightarrow v_2 \qquad \mathsf{APPLY}_{\mathsf{op}_2}(op_2, v_1, v_2) = v}{E \vdash atexp_1\ op_2\ atexp_2 \Rightarrow v} \tag{3.19}$$

$$\frac{E \vdash atexp \Rightarrow v}{E \vdash (atexp) \Rightarrow v} \tag{3.20}$$

We note that in (3.19), the same as in the static semantics, value identifier *vid* is looked up in the environment E.

Constructors $E \vdash constr \Rightarrow v/\mathsf{Fail}$

We present the inference rules of EAS constructors as follows.

$$\frac{E \vdash atexp_1 \Rightarrow v' \qquad \mathsf{APPLY}_{\mathsf{constr}}(constr, vid, v', atexp_2) = v}{E \vdash [vid : atexp_1 \mid atexp_2] \Rightarrow v} \tag{3.21}$$

$$\frac{E \vdash atexp_1 \Rightarrow v' \qquad \mathsf{APPLY}_{\mathsf{constr}}(constr, vid, v', atexp_2) = v}{E \vdash \{vid : atexp_1 \mid atexp_2\} \Rightarrow v} \tag{3.22}$$

We note that although the above inference rules (3.21) and (3.22) are not precise, this should cause no confusion for implementing the constructors.

Value Bindings $E \vdash bind \Rightarrow VE/\mathsf{Fail}$

The dynamic semantic inference rules of the EAS bindings are given as follows.

$$\frac{}{E \vdash \quad \Rightarrow \{\} \text{ in Env}} \tag{3.23}$$

$$\frac{E \vdash exp \Rightarrow v \qquad E, v \vdash vid \Rightarrow VE \qquad \langle E \vdash bind \Rightarrow VE' \rangle}{E \vdash vid = exp \Rightarrow VE \langle + \ VE' \rangle} \tag{3.24}$$

$$\frac{E \vdash exp \Rightarrow v \qquad E, v \vdash vid \Rightarrow \mathsf{Faul}}{E \vdash vid = exp \Rightarrow \mathsf{Fail}} \tag{3.25}$$

$$\frac{E \vdash bind \Rightarrow VE}{E \vdash bind \Rightarrow VE \text{ in Env}} \tag{3.26}$$

Expressions $C \vdash exp \Rightarrow v/\mathsf{Fail}$

The dynamic semantic inference rules of EAS expressions are given as follows.

$$\frac{E \vdash atexp \Rightarrow v}{E \vdash atexp \Rightarrow v} \tag{3.27}$$

$$\frac{E \vdash exp \Rightarrow b \qquad E \vdash atexp \Rightarrow v \qquad \mathsf{APPLY}_{\mathsf{bas}}(b, v) = v'}{E \vdash exp\ atexp \Rightarrow v'} \tag{3.28}$$

$$\frac{E \vdash bind \Rightarrow E' \qquad E + E' \vdash exp \Rightarrow v}{E \vdash \mathsf{let}\ bind\ \mathsf{in}\ exp\ \mathsf{end} \Rightarrow v} \tag{3.29}$$

$$\frac{E \vdash exp_1 \Rightarrow 0 \qquad E \vdash exp_3 \Rightarrow v}{E \vdash \mathsf{if}\ exp_1\ \mathsf{then}\ exp_2\ \mathsf{else}\ exp_3 \Rightarrow v} \tag{3.30}$$

$$\frac{E \vdash exp_1 \Rightarrow v'\ (v' \neq 0) \qquad E \vdash exp_2 \Rightarrow v}{E \vdash \mathsf{if}\ exp_1\ \mathsf{then}\ exp_2\ \mathsf{else}\ exp_3 \Rightarrow v} \tag{3.31}$$

In (3.29), notice that none of the rules for function application has a premise in which the operator evaluates to constructed value. This is because we are interested in the evaluation of well-typed programs only, and in such programs exp will always have a functional type.

3.3 EAS Dynamic Optimizer

A naive implementation of EAS programs makes a copy for a bound compound value in an updating expression. The EAS dynamic optimizer applies the dynamic

destructive effect analysis and the optimized program performs a destructive update for an updating expression at run time when it is safe.

The implementation of the EAS dynamic optimizer includes two phases:

- the enriched type system, which time stamps the program points and computes the last-use of variables, and
- the optimizing code generator, which augments the object code with the codes for the dynamic destructive effect analysis and destructive update optimization.

3.3.1 Enriched Type System

We apply the idea of Turner, Wadler and Mossin [TWM95], in which they developed a method for determining whether a value is used at most once based on a modification of the Hindley-Milner type system. We integrate the data collecting phase together with the type system of the EAS compiler.

In addition to type analysis, the enriched type system has two major tasks:

- giving each program point a time stamp, and
- computing the last-use of each variable.

The time stamp scheme we implemented is described in the appendix section of Chapter 2. We also implemented the comparing function for the use of the EAS

dynamic optimizer and the EAS program analyzer.

The semantic model of the variable last-use analysis is presented in Figure 2.1. The last use of a variable V is a set of the time stamps after any of which there is no reference to V.

3.3.2 Optimizing Code Generator

The EAS dynamic optimizer generates the code that implements the semantic functions of expressions for the dynamic destructive effect analysis shown in Figure 2.2. The basic idea is that when the EAS dynamic optimizer detects it is safe, it implements an updating expression with a destructive update.

In order for tracing back the destructive effects of values in the outer scopes at the time of function return, we adapted the structure effect μ of a value V in function f as

$$\mu_f(V) = \langle des,\ \Theta,\ \mu_g(V) \rangle$$

where, des is the interprocedural destructibility of V when f is called, Θ is the last-use of V in the execution of f, and $\mu_g(V)$ is the location of the destructive effect of V at the time when f is called. In such a way, when f returns a value V that is one of its arguments, we may use $\pi_3(\mu_f(V))$ to get the destructive effect when f is called.

Our optimizing code generator has two major tasks:

- it generates an optimizing version of each updating primitive, which makes a destructive update when it is safe and makes a fresh copy otherwise, and
- it compute the last-use of values along the program execution.

3.4 EAS Static Analyzer

The EAS static analyzer computes the destructive effects of expressions in a program at compile time. Since the computation assumes that the values in standard semantics are not available at compile time, the analyzer actually computes the destructive effects of the expressions of all the possible control flows.

The EAS static analyzer is an implementation of the abstract interpretation framework described in Section 2.5. We apply a two-level intermediate representation transformation to translate the abstract syntax tree of expressions into the representation in our abstract domain. Then, we compute the values of expressions in our abstract domain, each of which is a set of destructive effects. In particular, we compute the fixed points for the mutually recursive functions. At last, the analysis concludes for the cases of updating expressions whether it is safe for destructive update optimization or not.

3.4.1 Two-Level Intermediate Representation

In order to transform an abstract syntax tree of an EAS expression eas_exp into the representation in our abstract domain abs_exp, we apply a two-pass transformation as follows.

1. Transform the eas_exp expression into the intermediate representation int_exp. In this pass, we also initialize the values of the expressions of derived types as $\emptyset$; and the values of the expressions of base types as $\perp$. (The analysis for integer expressions is omitted.)

2. Transform the int_exp expression into the representation of our abstract domain abs_exp. In this pass, we compute all the possible destructive effects of each expression in three steps:

   - Compute the destructive effects of each abs_exp in the function definitions $f_1, \ldots, f_n$.
   - Solve the fixed points of mutually recursive functions.
   - Compute all the possible destructive effects of each abs_exp expressions in the main function of the program f_1. $\Box$

Please note that we use exactly the same code for the set operations of computing the expression values in the abstract domain, which are sets of destructive

effects, as that we generate as the object code for the set operations of the set-theoretic expressions in EAS programs.

3.4.2 Static Destructive Update Optimization

As soon as the destructive effects of the expressions are available, we may perform the static destructive update optimization. The applying method of the static destructive update optimization is straightforward:

> *It is safe to implement an updating expression with a destructive update when all the possible destructive effects of the expression indicate that the value is globally destructible.* □

More specifically, in the static destructive update analysis, we try to answer two kinds of questions based on the compile time analysis:

1. Is it safe to implement an updating expression in a program with a destructive update?

2. Is it safe to implement an updating expression with a destructive update for a certain occurrence along a specific control flow?

And, usually we give a more conservative answer for question 1 than that for question 2.

Consider the *swap* function in the bubble sort program in Figure 2.5. For question 1, it is not safe to implement the updating expression upd_1 with time stamp

2 by a destructive update, while it is safe to implement the updating expression upd_2 with time stamp 3 by a destructive update.

For question 2, let us consider the execution of $bubsort_2$ with time stamp 5 in the main function. Since b_4 is referred the last in the program, all the function calls to *swap* pass a dead value to *swap* as the first parameter, and therefore both upd_1 and upd_2 may be implemented by destructive updates for the execution of $bubsort_2$.

3.5 Benchmarks

We experiment on the performance of the naive implementation of programs and the dynamically optimized program. The naive implementation makes a fresh copy in each case of updates. The optimized programs include the code that performs a destructive update optimization at run time.

The programs that we experimented are:

- *BubbleSort*: a bubble sort that terminates when no swap occurs in an iteration,
- *HeapSort*: a heap sort for arrays in which the children of the i^{th} element are the $(2 \times i)^{th}$ and the $(2 \times i + 1)^{th}$ elements,
- *QuickSort*: a quick sort based on the divide-and-conquer paradigm, whose

expected running time is $O(n \log n)$ and worst-case running time is $O(n^2)$ on an array of n numbers,

- *TopoSort*2: a finite differencing version of topological sort that differentially updates the set of candidates,
- $Dijkstra$: Dijkstra's single-source shortest path algorithm implemented with a binary heap, and
- $Kruskal$: Kruskal's minimum spanning tree algorithm implemented with a priority queue by a binary heap.

We executed *BubbleSort*, *HeapSort*, and *QuickSort* on integer arrays of 100 numbers, and we executed *TopoSort*2, $Dijkstra$, and $Kruskal$ on graphs with 100 nodes and 300 edges. All the test data were automatically generated by random. We used the standard C library $\langle time.h \rangle$ to measure the cpu elapsed time [KR88], and translated the results into the numbers of seconds.[1] The empirical results are shown in Figure 3.10.

We computed the ratio of the optimized running time over the unoptimized running time. For example, the running time for the optimized $Kruskal$ program is 0.159 of the running time for the unoptimized $Kruskal$ program. In other words, 0.841 of the running time for the unoptimized $Kruskal$ program is spent

[1] The C functions for measuring benchmarks are coded by M. Nayakkankuppam for the SeQuL package.

Program	data size	unoptimized t_1 (sec)	optimized t_2 (sec)	$\frac{t_2}{t_1}$
BubbleSort	100	1.080	0.350	0.324
HeapSort	100	0.260	0.060	0.231
QuickSort	100	0.130	0.070	0.538
TopoSort2	100/300	0.110	0.070	0.636
Dijkstra	100/300	0.820	0.230	0.280
Kruskal	100/300	3.710	0.590	0.159

Figure 3.10: Benchmarks for the unoptimized and optimized programs

on unnecessary hidden copying. Note that the execution time of the optimized programs range from 0.159 to 0.636 of the execution time of the unoptimized programs.

Chapter 4

Finite Differencing Functional Programs

In this chapter we will be concerned principally with general sets and mappings represented by sets and tuples. The set-theoretic operations $S+T$, $S-T$, and $S*T$ compute the union set, difference set (set minus), and intersection set. Consider the program segments (a) and (b) in Figure 4.1. When the value bound to S is differentially changed and the value C of $S-T$ can be restored with either the expression of line 3 in (a) or the expression of line 3 in (b). The expected cost of computing $C = S - T$ is proportional to $\#S$, the cardinality of S, while $C = C + (\{x\} - T)$ can be computed in constant time. This leads to the study of the optimization technique of finite differencing.

Program segment (a)

```
1) C = S - T;
2) S = S + {x};
3) C = S - T;
```

Program segment (b)

```
1) C = S - T;
2) S = S + {x};
3) C = C + ({x} - T);
```

Figure 4.1: Program segments with set-theoretic expressions

Paige proposed four rules of finite differencing the set-theoretic expressions that are continuous in all of their parameters [Pai81]. We describe the rule 1 here as an example of showing how finite differencing works.

> (Finite differencing of set-theoretic expressions) Rule 1: We begin by making expressions with form $C = \{x \in S | K(x)\}$ available on the entrance to a program block B. Then, at each program point p inside B where the value of S changes by $S = S \pm A$, the value of C, which could be spoiled at p, is updated by inserting the pre-derivative code $C = C \pm \{x \in A | K(x)\}$. □

With this rule, the result of finite differencing of program segment (c) is shown as program segment (d)[1] in Figure 4.2.

[1] $C = C + \{x \in A | K(x)\}$ is transformed into the form of *if* $K(x)$ *then* $C = C + \{x\}$; *end if*;.

Program segment (c)

```
forall x in S
   if f(x) then
      S = S + {x};
   end if;
   C = { x in S | K(x) };
end forall;
```

Program segment (d)

```
C = { x in S | K(x) };
forall x in S
   if f(x) then
      S + {x};
      if K(x) then
         C = C + {x};
      end if;
   end if;
end forall;
```

Figure 4.2: Example of finite differencing of set-theoretic expressions

The formal specification of finite differencing is presented in terms of a collection of program transformations. The finite differencing transformed programs were shown to improve the execution performance from the original version of programs [PK82].

We briefly describe the finite differencing of computable expressions in the following section. In section 4.2, we present the way that we perform finite differencing transformations on functional set-theoretic expressions. We extend, in section 4.3, the application of finite differencing technique to the optimization of list expressions. The benchmarks are included in section 4.4.

4.1 Finite Differencing of Computable Expressions

Finite differencing of computable expressions was developed from Paige's technique of formal differentiation [Pai81] as a global program optimization method that captures a commonly occurring yet distinctive mechanism of program construction in which repeated costly calculations are replaced by inexpensive incremental counterparts. When finite differencing is applied to algorithms expressed as high-level, lucid, but inefficient program statements, the transformed algorithms materialize as more complex but efficient program versions. This method generalizes John Cocke's method of strength reduction, and provides a convenient framework for implementing a host of program transformations, including Earley's "iteration in-

version" [PK82].

The idea of finite differencing was originated with "reduction in strength." Reduction in strength is viewed as an extension of code motion whereby the major cost of evaluating an expression $E = f(x_1, \ldots, x_n)$ is moved outside a program region R despite modifications to its parameters $x_1, \ldots, x_n$ occurring within R. By making E available on the entry to R and keeping E available within R with appropriately modifying E each time one of the parameters $x_1, \ldots, x_n$ is modified, we can avoid full calculations of f within R by replacing the expression E for the redundant occurrences of f within R. For the case that the cost of keeping E available in R is less than the cost of calculating f anew each time it is referred, the finite differencing technique is valuable and useful.

Finite differencing is formulated in terms of three semantics preserving program transformations that generalize the above reduction in strength schema.

1. **Init transformation**: The Init transformation $Init(P)$ replaces each contiguous sequence of achieve statements, *e.g.* achieve $E = f(x_1, \ldots, x_n)$, within a program P by a code block B that computes and stores the values of the expressions $f(x_1, \ldots, x_n)$ into their respective virtual variable E.

2. **Differential transformation**: The Differentiation transformation $\partial J\langle R\rangle$ inserts code within a program region R in order to keep each expression $E = f(x_1, \ldots, x_n)$ belonging to a sequence of expressions J available at the

program points in R after which redundant uses of $f(x_1, \ldots, x_n)$ are replaced by E.

3. **Clean transformation**: The Clean transformation, which is the last step of finite differencing, eliminates redundant codes.

Finite differencing has been successfully applied to imperative language compilation and imperative program transformations. Our attempt is to extend the technique to functional language compilation and functional program optimization.

Among the three transformations, differential transformation is fundamental to finite differencing. We describe, in the section, the differential transformations and the *chain rule* formally specified by Paige [Pai81]. We also include the algorithm of detecting induction variable sets and the algorithm of applying the finite differencing transformations proposed by Paige [Pai81, PK82].

A differential transformation is defined in the following terms:

- an applicative expression $E = f(x_1, \ldots, x_n)$ where E is a variable uniquely associated with the value of $f(x_1, \ldots, x_n)$;
- a single-entry single-exit code block B that can modify the values of the variables $x_1, \ldots, x_n$ on which E depends; and
- a computable *derivative* ∂E that allows us to determine the new value E_{new} of E from its old value E_{old} when E_{old} is spoiled by changing the value with

dx_i to x_i on which E depends.

The computable derivative is formally defined as follows:

Definition 4.1 (Computable derivative): Let $E = f(x_1, \ldots, x_n)$ be an applicative expression that depends on the variables $x_1, \ldots, x_n$ and let dx_i be an expression changing the value of x_i. The code block pair $[B_1,\ B_2]$ is said to be a derivative of E with respective to dx_i if

1. the only variables modified by B_1 or B_2 are E and variables local to B_1 and B_2; and
2. the code block

   achieve $E = f(x_1, \ldots, x_n)$;

   B_1

   dx_i

   B_2

   assert $E = f(x_1, \ldots, x_n)$;

   preserves the semantics of dx_i and contains only redundant uses of $f(x_1, \ldots, x_n)$.

When $[B_1,\ B_2]$ is a derivative of E with respect to dx_i, we say that B_1 is a *pre-derivative* of E with respect to dx_i and that B_2 is a corre-

sponding *post-derivative* of E with respect to dx_i, for which we write $B_1 = \partial^- E\langle dx_i \rangle$, and $B_2 = \partial^+ E\langle dx_i \rangle$, respectively. Note that the occurrences of the variable x_i within B_1 refer to the old value of x_i prior to the change dx_i while those of x_i within B_2 refer to the new value $\Box$.

Note that when no uses of E within the derivative code B_1 or B_2 are live on the entry of B_1, we can omit the achieve statement in the code, in which case we that $[B_1, B_2]$ is a *strong* derivative.

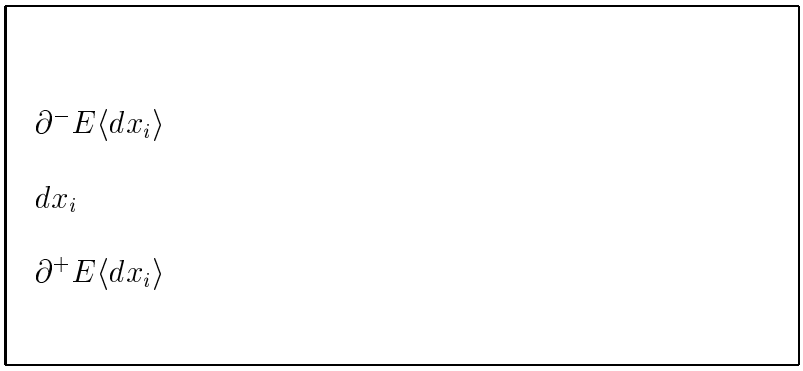
Now, we define *finite differencing* with *derivative code insertion* and *redundant evaluation elimination*.

Definition 4.2 (finite differencing): Consider an applicative expression $E = f(x_1, \ldots, x_n)$ that is well defined within a single-entry single-exit code block B occurring in a valid program P. Suppose that no uses of E in P are live within B. Suppose that the collection of derivative rules *Derives* are rules giving derivatives for E with respect to every assignment dx_i occurring in B to a variable x_i on which E depends. E is said to be *differentiable* with respect to B if

- f is well defined on entry to B, and
- the first expression of B is an updating expression with respect to which the derivative of E is a strong derivative.

The differential of E with respect to B, denoted as $\partial E\langle B\rangle$, is a new code block formed from B in the following ways:

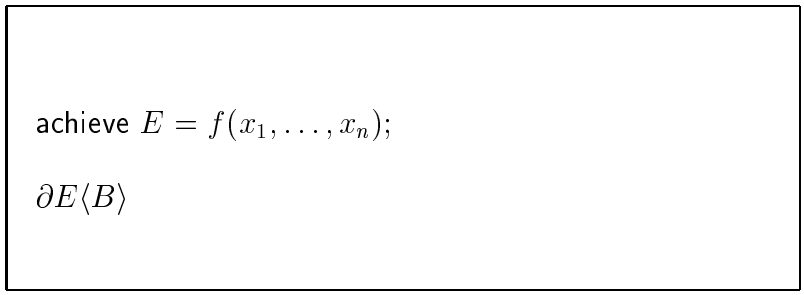
1. **Derivative Code Insertion**: Replace each statement dx_i, that modifies a variable x_i on which E depends by a code block



2. **Redundant Evaluation Elimination**: Replace all uses of $f(x_1, \ldots, x_n)$ within the code block by uses of the variable E. □

Now we are ready to introduce a theorem. Please refer to [PK82] for the detailed proof.

Theorem 4.3 (Semantic preservation) Let $E = f(x_1, \ldots, x_n)$ be an applicative expression that is differentiable with respect to a code block B in a valid program P. If any use of E occurring within $\partial E\langle B\rangle$ is live on the entry to $\partial E\langle B\rangle$, the code block



preserves the semantics of B; otherwise, $\partial E\langle B\rangle$ preserves the semantics of B. Furthermore, E is available on exit from $\partial E\langle B\rangle$. □

Corollary 4.4 (linearity of differential transformation) The *differential transformation* is a linear operator with respect to sequential blocks; that is, $\partial E\langle B_1\ B_2\rangle = \partial E\langle B_1\rangle \partial E\langle B_2\rangle$. □

The following definition and theorem will show how to differentiate sequences of expressions using a chain rule.

Definition 4.5 (Differential chain): Consider n expressions $E_1 = f_1$, ..., $E_n = f_n$ and a single-entry single-exit code block B occurring in a valid program P. Suppose that E_1 is differentiable with respect to B, that E_2 is differentiable with respect to $\partial E_1\langle B\rangle$, ..., and that E_n is differentiable with respect to $\partial E_{n-1}\langle \ldots \langle \partial E_1\langle B\rangle\rangle \ldots\rangle$. Suppose also that $i > j$ implies that f_j does not involve the variable E_i; i.e., $f_1, \ldots, f_n$ preserve an inner-to-outer subexpression ordering. Then, the list of virtual variables $E_n, \ldots, E_1$ is said to form a *differentiable chain*. And,

the extended differential of the chain $E_n, \ldots, E_1$with respect to B is defined recursively by the following "chain rule":

$$\partial E_n, E_{n-1}, \ldots, E_1\langle B\rangle = \partial E_n, E_{n-1}, \ldots, E_2\langle \partial E_1\langle B\rangle\rangle \Box \qquad (4.1)$$

It is important to further restrict the chain ordering (4.1) to prevent the derivative code for E_i from introducing any expression f_j, $j < i$, since such an occurrence of f_j might not be eliminated as redundant within the differential.

Theorem 4.6 (Chain Rule): Let $E_n = f_n, \ldots, E_1 = f_1$ be a chain of n applicative expressions differentiable with respect to a code block B occurring within a valid program P. Let S be the set of indices $i = 1, \ldots, n$ for which there are uses of E_i within $B' = \partial E_n, \ldots, E_1\langle B\rangle$ that are live on the entry to B'. Then the code block

achieve $\bigwedge_{i\in S} E_i = f_i$;

$\partial E_n, \ldots, E_1\langle B\rangle$

preserves the semantics of B and keeps $E_1, \ldots, E_n$ available on exit. Furthermore, if g is any expression formed from some f_j, $j = 1, \ldots, n$, by substituting f_i, for E_i, $i = 1, \ldots, j-1$. Then, any use of g occurring within B or introduced within derivative code by the chain rule will be

made redundant and therefore replaced by E_j within $\partial E_n, \ldots, E_1\langle B\rangle$.

□

Corollary 4.7(Linearity to blocks): The extended differential is a linear operator with respect to sequential code blocks; that is,

$$\partial E_n, \ldots, E_1\langle B_1,\ B_2\rangle = \partial E_n, \ldots, E_1\langle B_1\rangle \partial E_n, \ldots, E_1\langle B_2\rangle \square$$

In general, we must allow a different set of induction variables for every component of every elementary expression. We define induction sets as follows.

Definition 4.8 (Induction variable set): Given a variable v found in a parse tree of a loop L and an elementary function $f(x_1, \ldots, x_n)$, we say that v belongs in the i^{th} induction set for f, denoted as $IV(x_i, f)$, if the following two conditions hold:

1. All definitions of v in L in which finite differencing is applied match the parameter definition patterns in $D(x_i, f)$, and
2. For each such definition pattern the corresponding derivatives must consist of easy calculations relative to f. □

We now give the details of the algorithms used for detecting reduction candidate expressions in Figure 4.3. We use an auxiliary function $Expand(f, Pfunc)$ for macro expansion. The two parameters of $Expand$ are a function form f and a map

- Input: a derivative table D, a set of elementary form F, A parse tree L of the optimization loop, and a map $Defs$ which associates each variable name v occurring in L with the set $Defs(v)$ of nodes in L corresponding to statements which can modify the value of v.
- Output: a loop L that is a candidate expression for finite differencing optimization.

1. Find the set RC of nodes in L corresponding to region constant expressions of L.
2. Compute initial sets $IV(x, f)$ of induction variables for every elementary form $f \in F$ and each pattern variable x of f.
3. Initialize Prologue to an empty code block.
4. While $\exists$ a node $t \in L$ and an elementary form $f(x_1, \ldots, x_n) \in F$ such that
   (a) $Match(t, f, Pfunc)$, and
   (b) for $i = 1, \ldots, n$, either $Pfunc(x_i) \in RC$ or $\overline{x_i} \in IV(x_i, f)$,

   perform finite differencing optimization algorithm. Finally, eliminate redundant code, and halt.

Figure 4.3: Algorithm detecting reduction candidate expressions

$Pfunc$ which associates each formal parameter x of f with a parse tree $Pfunc(x)$. $Expand(f, Pfunc)$ returns the root of a new parse tree which results from replacing each formal parameter x of f by $Pfunc(x)$. The predicate $Match(t, f, Pfunc)$ indicates whether f matches a tree t. If $Match(t, f, Pfunc)$ holds, we use the symbol $\overline{f(\overline{x_1}, \ldots, \overline{x_n})}$ as an abbreviation for $Text(t)$, and we use the term $\overline{f}(y_1, \ldots, y_n)$ to express $Text(Expand(f, Pfunc))$, where $Pfunc(x_i) = t_i$, and $y_i = Text(t_i)$, $i = 1, \ldots, n$.

The algorithm of finite differencing optimization is presented in Figure 4.4. We assume that before the finite differencing transformations are applied the code

structure is in the form of a parse tree, over which a control flow graph is imposed. Data flow analysis is worked out so that the $usetodef$ and $deftouse$ maps are defined. Type analysis is also performed.

We now sketch an algorithm that could actually automate all of the transformational steps. Certain implementation-level details are absent here, but may be found in [PK82]. The automatic finite differencing algorithm is shown in Figure 4.5.

Finite differencing of applicative expressions extends an old mathematical idea to the general problem of algorithm optimization and to the implementation of high-level languages. The technique of finite differencing offers new and efficient implementations of very high-level programming language dictions. Our goal is to extend the application of finite differencing to functional language implementations.

4.2 Finite Differencing Functional Set-Theoretic Programs

We present our extension of the finite differencing technique to functional programs with set-theoretic expressions. Since in functional programs the control flows are in the form of recursive function calls, we need a new algorithm to compute the sets of induction variables. Also, we detect three groups of candidate expressions for finite differencing, namely unary differentiable expressions, binary differential

- Input: a derivative table D, a set of elementary form F, A parse tree L of the loop detected as a reduction candidate, and a map $Defs$ which associates each variable name v occurring in L with the set $Defs(v)$ of nodes in L corresponding to statements which can modify the value of v.
- Output: a new optimized loop L' and its prologue code block.

1. Generate a unique variable $v_{\overline{f}}$ for keeping the matched expression $\overline{f}$ available in L, and insert an assignment $v_{\overline{f}} := \overline{f}$ at the end of the Prologue block.
2. For each expression $\overline{x_i}$, $i = 1, \ldots, n$, such that $\overline{x_i} \in IV(x, f)$, and for each program point $p \in Defs(\overline{x})$ at which $\overline{x}$ undergoes a change $\overline{x} = \Delta_{\overline{x}}$, insert appropriate derivative code can be generated by first finding the unique triple $\langle mod, preD, postD \rangle$ belonging to $D(x_i, f)$ in which $Match(p, mod, Pfunc)$ holds. Next, to prepare for macro expansion we must produce a new pattern variable map $Sfunc$ (which can be formed from $Pfunc$) which maps pattern variables found in $preD$ and $postD$ into appropriate trees. Finally, we expand the pre-derivative and post-derivative patterns $preD$ and $postD$ by executing $Expand(preD, Sfunc)$ and $Expand(postD, Sfunc)$, and insert the resulting code immediately before and after p.
3. Within L replace all occurrences of $\overline{f}$ by $v_{\overline{f}}$. Also, within the derivative code generated in step 6 substitute the variable v_e for any expression e which has already been reduced. Finally, make appropriate additions to the set RC of region constants and to the induction sets IV.

Figure 4.4: Algorithm for finite differencing optimization

Algorithm: Automatic Finite Differencing

1. Apply preparatory transformations.
2. Decompose the program into its loop structure $L_1, \ldots, L_n$ with the property $i < j \Rightarrow L_i$ is contained in L_j or $L_i \cap L_j = \{\}$.
3. For $i = 1 \ldots n$ determine a chain J_i of differentiable expressions to be reduced within L_i, but not in any region enclosing L_i.
4. For $i = n, n-1, \ldots, 1$ transform L_i into
   achieve $\bigwedge_{(E=f)\in J} E = f$;
   $\partial J_i \langle L_i \rangle$.
5. If P is the program that results from step 4, transform P into the final program $Clean\langle Init\langle P \rangle\rangle$.

Figure 4.5: Automatic finite differencing algorithm

expressions, and implicit binary differentiable expressions.

After we detect the induction variables and the candidate expressions for finite differencing, we may apply the algorithm for finite differencing optimization in Figure 4.4 and the automatic finite differencing algorithm in Figure 4.5 to optimize the set-theoretic expressions in functional programs.

4.2.1 Induction Variable

In functional programming paradigm loops are in a form of recursive function calls, and the induction variables appear as function parameters. This leads to the need of a new algorithm for identifying the sets of induction variables when the finite

```
fun foo1 S T =
    if isemp S then T;
        else let a = rep S;
                 Q = S less a;
                 R = {x in Q | k1(x)};
             in foo1 Q {y in R | k2(y)};
```

Figure 4.6: Example of identifying induction variables in a functional program

differencing transformations are applied to functional programs.

As an example, in Figure 4.6, the first parameter S of function $foo1$ is an induction variable while the second parameter T is not.

Since we are interested in applying the finite differencing technique to the expressions in functional programs which actually perform as loops in the control flow of the programs, we propose the following algorithm for detecting the induction variables of candidate differentiable expressions.

Definition 4.9 (induction variable set of recursive functions): Given a variable v found in a parse tree of a loop L and an elementary function $f(x_1, \ldots, x_n)$, we say that v belongs in the i^{th} induction set for f, denoted as $IV(x_i, f)$, if the following three conditions hold:

1. In the definition of f, the formal parameter x_i appears in the predicate of an if expression of which one arm includes recursive calls to f and the other does not,

2. All definitions of v in L in which finite differencing is applied match parameter definition patterns in $D(x_i, f)$, and

3. For each such definition pattern the corresponding derivatives must consist of easy calculations relative to f. □

4.2.2 Unary Differentiable Expressions

Unary differentiable expressions are the set-theoretic expressions that are differentiable with respect to one of their parameters. Consider the set former expression

$$E_1 = \{x \in S | k(x)\}.$$

The differential forms of E_1 with respect to the changes S with y and S less y are

```
if k(y) then
   E1 with y;
end if;
```

and

```
if k(y) then
   E1 less y;
end if;
```

respectively.

For example, in Figure 4.7 we transform the unary differentiable expression in program segment (e) into the differential form in program segment (f).

Please note that in the case that the implementation of sets does not include the cardinality, the cardinality expression $\#S$ is also a unary differentiable expression.

Program segment (e)

```
fun f2 ( S, T, E ) =
   if isemp S then E
   else let y = rep S;
          in if y > 20 then let R = T less y;
                             in f2(S less y, R, { x in R | x > 0 })
             else f2(S less y, T, { x in T | x > 0 });

fun f ( S ) = f2(S, S, { x in S| x > 0 });
```

Program segment (f)

```
fun f2 ( S, T, E ) =
   if isemp S then E
   else let y = rep S;
          in if y > 20 then f2(S less y, T less y, E less y)
             else f2(S less y, T, E);

fun f ( S ) = f2(S, S, { x in S| x > 0 });
```

Figure 4.7: A unary differentiable expression

4.2.3 Binary Differentiable Expressions

Binary differentiable expressions are the set-theoretic expressions that are differentiable with respect to two of their parameters. Consider the following set expressions

$$\begin{aligned} E_2 &= S + T, \\ E_3 &= S * T, \text{ and} \\ E_4 &= S - T. \end{aligned}$$

The differential forms of E_2 with respect to the changes S with y and S less y are

```
if !(y in T) then
   E2 with y;
end if;
```

and

```
if !(y in T) then
   E2 less y;
end if;
```

respectively. We may derive the differential forms of E_3 and E_4 with respect to the changes S with y and S less y in a similar way.

For example, in Figure 4.8 we transform the binary differentiable expression in program segment (g) into the differential form in program segment (h).

Program segment (g)

```
fun g2 ( R, S, T, E ) =
   if isemp R then E
   else let y = rep R;
          in if y > 20 then let Q = S less y;
                                in g2(R less y, Q, T, Q+T)
             else g2(R less y, S, T, S+T);

fun g ( S, T ) = g2(S, S, T, S+T);
```

Program segment (h)

```
fun g2 ( R, S, T, E ) =
   if isemp R then E
   else let y = rep R;
          in if y > 20 then
                if y in T then g2(R less y, S less y, T, E)
                else g2(R less y, S less y, T, E less y)
             else g2(R less y, S, T, E);

fun g ( S, T ) = g2(S, S, T, S+T);
```

Figure 4.8: A binary differentiable expression

4.2.4 Implicit Binary Differentiable Expressions

Consider E_6 in the following code.

$$\begin{aligned} E_5 &= S + T; \\ E_6 &= \{x \in E_5 | k(x)\}; \end{aligned}$$

We say E_6 is an implicit binary differentiable expression in S and T.

We apply the systematic scheme described in Section 4.1 to compute the finite differencing of the implicit binary differentiable expressions. The differential form of E_6 with respect to the change S with y is

$$\begin{aligned} \partial E_6, E_5\langle S \text{ with } y\rangle \;\rightarrow\; & \partial E_6\langle \text{if } y \notin T \text{ then} \\ & \quad E_5 \text{ with } y; \\ & \text{end if}; \\ & S \text{ with } y;\rangle \\ \rightarrow\; & \langle \text{if } y \notin T \text{ then} \\ & \text{if } k(y) \text{ then} \\ & \quad E_6 \text{ with } y; \\ & \text{end if}; \\ & \text{end if}; \\ & S \text{ with } y;\rangle \end{aligned}$$

Program segment (i)

```
fun h2 ( R, S, T, E ) =
   if isemp R then E
   else let y = rep R;
          in if y > 20 then let Q = S less y;
                             in h2(R less y, Q, T, {x in Q++T| x>0})
             else h2(R less y, S, T, {x in S++T| x>0});

fun h ( S, T ) = h2(S, S, T, {x in S++T| x>0});
```

Program segment (j)

```
fun h2 ( R, S, T, E ) =
   if isemp R then E
   else let y = rep R;
          in if y > 20 then
                if y in T then h2(R less y, S less y, T, E)
                else h2(R less y, S less y, T, E less y)
             else h2(R less x, S, T, E);

fun h ( S, T ) = h2(S, S, T, {x in S++T| x>0});
```

Figure 4.9: An implicit binary differentiable expression

We may derive the differential forms of E_6 with respect to the change S less y in a similar way.

For example, in Figure 4.9 we transform the implicit binary differentiable expression in program segment (i) into the differential form in program segment (j).

A naive version:

```
1.  fun TSort ( S, T, L ) =
2.      if isemp T then L;
3.         else let a = rep T;
4.                  S1 = S less a;
5.              in TSort( {a in S1 | sp{a}**S1=={} }, S1, a::L );

6.  fun TopSort S =
7.      TSort( {a in S | sp{a} ** S == {} }, S, [] );
```

Figure 4.10: A topological sort program in EAS

4.2.5 Example

Consider the topological sort program in Figure 4.10. We detect the function $TSort$ as a candidate expression for finite differencing with respect to the induction variable S. We note that the recursive call to $TSort$ repeatedly construct the set $\{a \in S_1 | sp\{a\} \cap S_1 == \emptyset\}$.

We present the finite differencing optimized program version in Figure 4.11. We apply a few finite differencing transformations to the program.

1. We replace the intersection operation $S = sp\{a\} \cap S_1$ with respect to S_1 less x by updating S with its differential form. Therefore, we may transform the predicate expression $sp\{a\} \cap S_1 == \emptyset$ into testing if the cardinality of S, or $\#S$, is 0. This leads to the introduction of map $Count$.

2. The pre-derivative code $\partial^{-} TSort$ is derived as a function $initTS$.

A finite differencing optimized version:

```
 1.  fun initTS ( sp, domS, S, C ) =
 2.      if isemp sp then <S, C>;
 3.         else let <x, y> = rep sp;
 4.                in if y in domS
 5.                   then let z = S{y};
 6.                            c = C{y};
 7.                        in initTS( sp less <x, y>, domS,
     S with <y, z with x>, C with <y, c+1>);
 8.                   else initTS( sp less <x, y>, domS with y,
     S with <y, {x}>, C with <y, 1>);

 9.  fun chkZ ( S, Z, C ) =
10.      if isemp S then <Z, C>;
11.         else let u = rep S;
12.                  c = C{u};
13.                in if c == 1 then chkZ( S less u, Z with u,
     C with <u,0> );
14.                      else chkZ( S less u, Z, C with <u, c-1> );

15.  fun TSort ( Z , S, C, L ) =
16.      if isemp Z then L;
17.         else let a = rep Z;
18.                  <Z1, C1> = chkZ( S{a}, Z, C );
19.                in TSort( Z1, S, C1, a::L );

20.  fun TopSort S =
21.      let <Prec, Count> = initTS( sp, {}, {}, {} );
22.          Z = { a in S | Count{a} == 0 };
23.       in TSort( Z, Prec, Count, [] );
```

Figure 4.11: A finite differencing optimized topological sort program

3. We transform the repeated construction of $\{a \in S_1 | sp\{a\} \cap S_1 == \emptyset\}$ into its differential form, which is the expressions needed for maintaining the set Z in function $chkZ$.

4. After removing the redundant evaluations, nothing is needed for the post-derivative code $\partial^+ TSort$.

4.3 Finite Differencing General Functional Programs

The technique of finite differencing may be applied to general functional expressions as well besides the set-theoretic expressions. One of the examples is the list constructor expression $[x : L | k(x)]$, which is a unary differentiable expression.

The calculation of the list constructor expression

$$E_7 = [x : L | k(x)]$$

is actually performed by the following standard function foo.

```
fun foo [] = [];
  | foo y::L =
    if k(y) then y::(foo L);
    else foo L;
```

Let $L = y :: L_t$, the differential form for $E_7(L) = [x : L|k(x)]$ relative to the changes tl L is

```
if k(y) then
   y :: tl E7(Lt);
else
   E7(Lt);
end if;
```

For example, in Figure 4.12 we transform the differentiable list expression in program segment (k) into the differential form in program segment (l).

4.4 Benchmarks

We measured the running time of a topological sort program to show the effectiveness of a finite differencing optimization [2]. We experimented for four cases:

1. Run the program compiled with no optimization at all,

2. Run the program compiled with a finite differencing optimization only,

3. Run the program compiled with a destructive update optimization only, and

[2] More empirical results are coming soon.

Program segment (k)

```
fun p ( L ) =
   if isnil L then ([], [])
   else let y = hd L;
            M = fst p(tl L);
            if y>20 then (M, [ x : M | x>0 ])
            else (y::M, [ x : (y::M) | x>0 ]);
```

Program segment (l)

```
fun p ( L ) =
   if isnil L then ([], [])
   else let y = hd L;
            A = p(tl L);
            if y>20 then A
            else if y>0 then (y::fst A, y::snd A)
                 else (y::fst A, snd A);
```

Figure 4.12: A unary differentiable list expression

Program	unoptimized t_1 (sec)	optimized t_2 (sec)	$\frac{t_2}{t_1}$
TopoSort1	0.960	0.970	1.010
TopoSort2	0.110	0.070	0.636
$\frac{\text{TopoSort2}}{\text{TopoSort1}}$	0.1146	0.0722	0.0729*

* is $\frac{\text{optimized TopoSort2}}{\text{unoptimized TopoSort1}}$.

Figure 4.13: Execution time of the programs

4. Run the program compiled with both finite differencing optimization and destructive update optimization.

The empirical results are shown in Figure 4.13.

The program *TopoSort*1 is a topological sort that repeatedly uses the set constructor $\{a \in S | sp\{a\} \cap S = \emptyset\}$ to compute the set of candidates. Since the cost of constructing a set anew is high, we apply finite differencing and transform the expression into its differential form. Thus, when we have the set available at the entry of a code block or a function, we may keep the set updated with the differential expressions which keep the set updated step by step. The program *TopoSort*2 is the program transformed from *TopoSort*1 with a finite differencing optimization.

We ran the programs *TopoSort*1 and *TopoSort*2 on graphs with 100 nodes and 300 edges. The test data is automatically generated by random. We used the standard C library [KR88] for measuring the time, and the results are translated into number of seconds.

We note the following:

- The running time of unoptimized *TopoSort*2 is 0.1146 of the running time of unoptimized *TopoSort*1, which shows that the unoptimized *TopoSort*1 program spent much of its running time on repeatedly constructing the set of candidates. The running time be substantially reduced by finite differencing transformations. If the program is compiled by an optimizing compiler with dynamic destructive update optimization, we may further reduce the running time. More precisely, the running time of the optimized *TopoSort*2 program is 0.0729 of the running time for the optimized *TopoSort*1 program.

- The running time for the optimized *TopoSort*1 program is slightly more than the running time for the unoptimized *TopoSort*1 program, which shows that in the case of no hidden copy additional time is spent on dynamic destructive update optimization. For *TopoSort*1 the additional time for a dynamic destructive update optimization is 0.010 of the total running time.

- The running time for the optimized *TopoSort*2 program is 0.636 of the running time for the unoptimized *TopoSort*2 program, which shows the effectiveness of a destructive update optimization applied together with finite differencing.

Chapter 5

Conclusion and Future Directions

We recapitulate in this chapter the results developed in this dissertation. We also contemplate several directions of future work.

5.1 Destructive Effect Analysis for a Fist-Order Language

We have developed a semantic model of destructive effect analysis and its corresponding abstract interpretation for a destructive update optimization of first-order strict functional languages. It is shown that our algorithms are quite effective when compared with the algorithms of some well-known methods.

Our method uses the time stamps for the liveness analysis. From the liveness of a value, we analyze the local destructibility, the interprocedural destructibility, and the global destructibility of the value. For a destructive update optimization,

we implement an updating expression with a destructive update when it is safe according to our analysis; in other words, when the value is globally destructible.

We implemented an optimizer with the dynamic destructive effect analysis. The optimizer computes the last-use of variables in the phase of elaboration, and generates optimizing code for a destructive update optimization at run time.

We also implemented a static analyzer of the destructive effect analysis. The analyzer computes the set of destructive effects for all possible values of each expression. We may analyze a program and decide at compile time whether an updating expression may be implemented with a destructive update.

The finite differencing optimization shows a good motivation and application of our analysis. When a destructive update analysis is available, transforming a repeatedly costly expression into its differential counterparts may improve the performance of a program. We modified the algorithms of detecting induction variables and reduction candidate expressions so that the finite differencing optimization may be practically applied to functional programs.

5.2 Destructive Effect Analysis for a Higher-Order Language

One of our future directions is extending the destructive effect analysis to higher-order languages. We found that a straightforward extension to the destructive

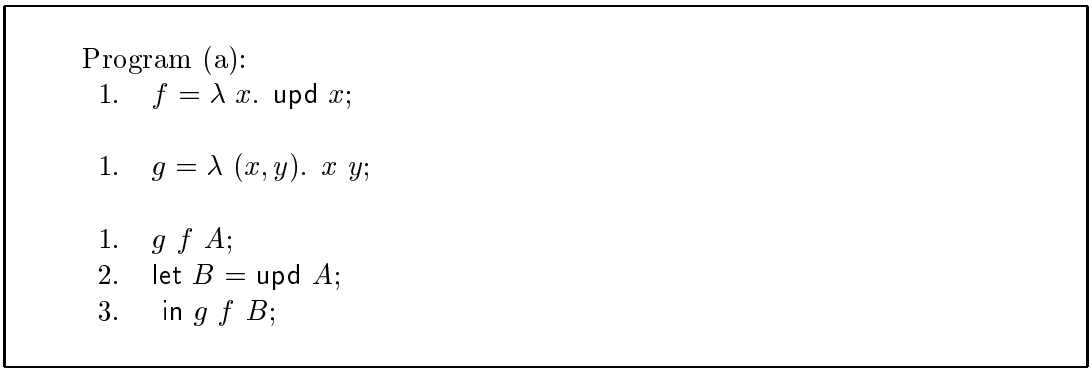


Figure 5.1: A simple higher-order program

effect analysis may be applied to a few of the higher-order cases.

For example, let us consider the program a in Figure 5.1, and apply the destructive effect analysis. The last-use of A is $\{2\}$, and the last-use of B is $\{3\}$. The destructive effect analysis is as follows.

- For the expression of time stamp 1, since A is not locally destructible ($1 \notin \{2\}$), the interprocedural destructibility is $false$ when A is passed into function g as the second parameter. For the function application $x\ y$ in the execution of g, the interprocedural of y is $false$ when y is passed into x, or function f. Hence, the upd expression in the execution of function f is implemented with making a fresh copy since the global destructibility of the value bound to x is $false$.

- For the expression of time stamp 2, since $2 \in \{2\}$, the global destructibility of the value bound to A is $true$, and we may perform a destructive update

on A.

- For the expression of time stamp 3, since $3 \in \{3\}$, the interprocedural destructibility of B is *true* when B is passed into function g as the second parameter. For the function application $x\ y$ in the execution of g, the interprocedural of y is *true* when y is passed into x, or function f. Hence, the upd expression in the execution of function f is implemented with a destructive update since the global destructibility of the value bound to x is *true*. □

However, for some more complicated programs, the straightforward extension of the destructive effect analysis could not catch the updating cases and analyze for a destructive update optimization. For example, function p in Figure 5.2 takes a function as its parameter x and apply the function x with A as the argument. Since each application of function p will actually refer to A, the destructive update analysis needs not only the global destructibility of A but also the global destructibility of p so that the analysis can decide whether it is safe for a destructive update or not.

We envision that a destructive effect analysis for higher-order languages may be derived with the analysis for the global destructibility of the functions whose definition integrates with aggregates, which are candidates for a destructive update optimization. More precisely, we are interested in analyzing the destructive effects of a function class $\mathcal{F}$. A function $f \in \mathcal{F}$ iff f is derived from applying a higher-order

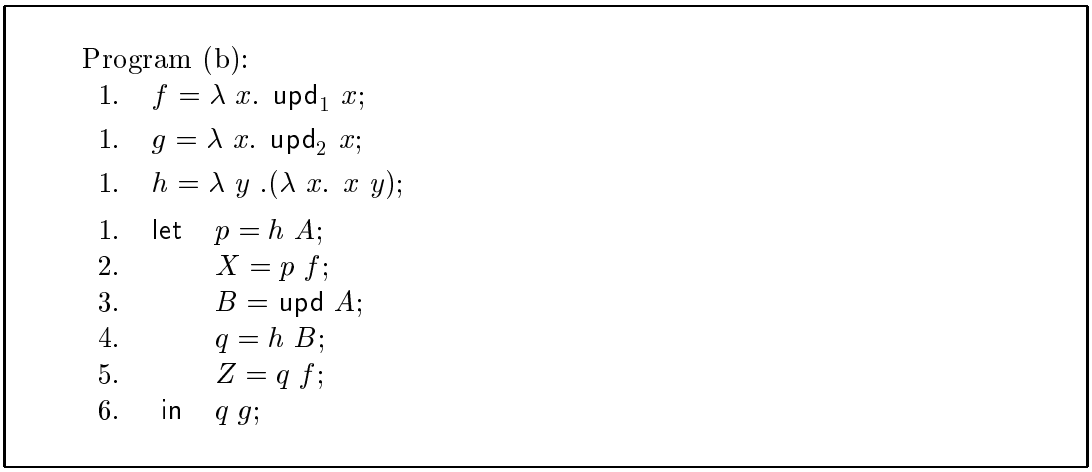


Figure 5.2: The destructibility of higher-order functions

function g to aggregate values $V_1, \ldots, V_n$.

In such a way, the destructive effect analysis for the program b in Figure 5.2 is as follows.

- The last-use of A is $\{3\}$, the last-use of B is $\{4\}$, the last-use of p is $\{2\}$, and the last-use of q is $\{6\}$.

- The application of h to A at time stamp 1 actually extends the liveness of the value V_A bound to A such that V_A is live iff either p or A is live. After binding $h\ A$ to p, the last-use of V_A is $\{3\}$.

- Similarly, after binding $h\ B$ to q at time stamp 4, the last-use of V_B is $6\}$.

- The function application $p\ f$ at time stamp 2 could not destructively update V_A since V_A is not globally destructible ($2 \notin \{3\}$).

- The upd expression at time stamp 3 is implemented with a destructive update since the global destructibility of V_A is $true$.

- in the execution of function q evoked with $q\ f$ at time stamp 5 could not destructively update V_B since V_B is not globally destructible.

- in the execution of function q evoked with $q\ g$ at time stamp 6 may be implemented by a destructively update on V_B since the global destructibility of V_B is $true$ ($6 \in \{6\}$). $\square$

Bibliography

[ASU86] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers Principles, Techniques, and Tools*, Addison-Wesley Publishing, 1986.

[AHU74] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *The Design and Analysis of Computer Algorithms*, Addison-Wesley Publishing, 1974.

[App92] A. W. Appel. *Compiling with Continuations*, Cambridge University Press, 1992.

[AR95] S. Anglade, and G. Richard. "S3L: a Fair Functional Language Implementing Infinite Sets," appeared in TENCON '91, revised on November 1995.

[AN87] Arvind and R. S. Nikhil. "Executing a Program on the MIT Tagged-Token Dataflow Architecture," *PARLE: Parallel Architectures and Languages Europe 1987*, Volume II: Parallel Languages, pp. 1-29, Netherlands, June 1987.

[ANK87] Arvind, R. S. Nikhil, and K. P. Keshav. "I-structures: data structures for parallel computing," *Proceedings of the Workshop on Graph Reduction*, New Mexico, February 1987.

[ACG96] I. Attali, D. Caromel, R. Guilder and A. L. Weldelborn. "Optimizing Sisal Programs: a Formal Approach," *Proceedings of International Conference on Parallel Processing*, LNCS 1123-1124, Springer-Verlag, Lyon, August 1996.

[ACW96] I. Attali, D. Caromel and A. L. Weldelborn. "A Formal Semantics and an Interactive Environment for Sisal," *Tools and Environments for Parallel and Distributed Systems*, edited by A. Zaky and T. Lewis, Kluwer Academic Publishers, February 1996.

[ACC95] I. Attali, D. Caromel, Y-S. Chen, R. Guilder and A. L. Weldelborn. "A Formal Semantics for Sisal Arrays," *Proceedings of Joint Conference on Information Information Sciences (JCIS]95)*, North Carolina, October 1995.

[Ban97] A. Banerjee. "A Modular, Polyvariant and Type-based Closure Analysis," *Proceedings of second ACM International Conference on Functional Programming*, pp. 1-10, The Netherlands, June 1997.

[Ber82] F. Le Berre. *Un Langage pour manipuler les ensembles: MANENS*, Ph.D. thesis, Paris VII, 1982.

[Blo94] A. Bloss. "Path Analysis and the Optimization of Non-strict Functional Languages," *ACM Transactions on Programming Languages and Systems*, Vol. 16, No. 3, pp. 328-369, May 1994.

[Blo89a] A. Bloss. *Path Analysis and the Optimization of Non-strict Functional Languages*, Ph.D. thesis, YALEU/DCS/RR-704, Yale University, May 1989.

[Blo89b] A. Bloss. "Update Analysis and the Efficient Implementation of Functional Aggregates," 4^{th} *International Conference on Functional Programming and Computer Architecture*, pp. 26-38, 1989.

[BCF92] J. M. Boyle, M. Clint, S. Fitzpatrick and T. J. Harmer. "Deriving DAP Implementations of Numerical Mathematical Software through Automated program Transformation," *Technical Report* 1992/Jul-JMB.MC.SF.TJH, Computer Science Department, The Queen's University of Belfast, July 1992.

[BH91] J. M. Boyle and T. J. Harmer. "Functional Specifications for Mathematical Computations," *Constructing Programs from Specifications*, edited by B. Moller, North-Holland Press, pp. 205-242, May 1991.

[Boy89] J. M. Boyle. "Abstract Programming and program transformations – An approach to reusing programs," *Software Reusability*, Vol. 1, edited by T. J. Biggerstaff and A. J. Perlis, pp. 361-413, ACM Press, 1989.

[BM86] J. M. Boyle and M. N. Muralidharan. "Program Reusability through Program Transformation," *Tutorial: Software Reusability*, edited by P. Freeman, pp. 235-249, IEEE the Computer Society Press, December 1986.

[CP93] J. Cai, and R. Paige. "Towards Increased Productivity of Algorithm Implementation," *Proceedings on ACM SIGSOFT 1993*, pp. 71-78, also in *ACM Software Engineering Notes*, Vol. 18, No. 5, December 1993.

[CE95] D. C. Cann and P. Evripidou. "Advanced Array Optimizations for High Performance Functional languages," *IEEE Transactions on Parallel and Distributed Computing*, Vol. 6, No. 3, pp. 229-239, March 1995.

[CBC93] J.-D. Choi, M. Burke, and P. Carini. "Efficient Flow-Sensitive Interprocedural Computation of Pointer-Induced Aliases and Side Effects," *Proceedings of* 20^{th} *ACM Symposium on Principles of Programming Languages*, pp. 232-245, Jan. 1993.

[CG92] T-R. Chuang, and B. Goldberg, "Backward Analysis for Higher-Order

Functions Using Inverse Images," *Technical Report*, TR1992-620, Computer Science Department, New York University, November 1992.

[CLR90] T. H. Cormen, C. E. Leiserson, and R. L. Riverst. *Introduction to Algorithms*, The MIT Press, 1990.

[DP93] M. Draghicescu, and S. Purushothaman. "A Uniform Treatment of Order of Evaluation and Aggregate Update," *Theoretical Computer Science*, B , 2(118), September 1993; also in *Proceeding of the 1990 ACM Conference on Lisp and Functional Programming.*

[FCO90] J. T. Feo, D. C. Cann, and R. R. Oldehoeft. "A Report on the Sisal Language Project," *Journal of Parallel and Distributed Computing 1990*, Vol. 10, No. 4, pp. 349-366, December 1990.

[FO95] S. M. Fitzgerald, and R. R. Oldehoeft. "Update-in-place Analysis for True Multidimensional Arrays," *High Performance Functional Computing*, A. P. Wim Bohm, and J. T. Feo, editors, pp. 105-118, April 1995.

[FSS83] S. M. Freudenberger, J. T. Schwartz, and M. Sharir. "Experience with the SETL Optimizer," *ACM Transaction on Programming Languages and Systems*, pp. 26-45, Vol. 5, No. 1, January 1983.

[GH89] K. Gopinath, and J. L. Hennessy. "Copy Elimination in Functional

Languages," 16^{th} *ACM Symposium on Principles of Programming Languages*, pp. 303-314, 1989.

[Gou94] J. Goubault. "HimML: Standard ML with fast sets and maps," *ACM SIGPLAN Workshop on Standard ML and its Applications*, June 1994.

[GP98] D. Goyal, and R. Paige. "A New Solution to the Hidden Copy Problem," *Technical Report #763*, Computer Science Department, New York University, New York, 1998.

[Hud87] P. Hudak. "A Semantic Model of Reference Counting and Its Abstraction," *Abstract Interpretation of Declarative Languages*, S. Abramsky and C. Hankin, editors, Ellis Horwood Press, 1987.

[HB85] P. Hudak, and A. Bloss. "The Aggregate Update Problem in Functional Programming Systems," 12^{th} *ACM Symposium on Principles of Programming Languages*, pp. 300-314, 1985.

[Joh85] T. Johnsson. "Lambda Lifting: Transforming Programs to Recursive Equations," *Functional Languages and Computer Architecture*, LNCS Vol. 201, pp. 190-203, Springer-Verlag Press 1985.

[KR88] B. W. Kernighan, and D. M. Ritchie. *The C Programming Language*, second edition, Prentice-Hall Press, 1988.

[Lac92] J.-J. Lacrampe. "S3L à tire d'ailes," Technical Report 92-11, Laboratoire d'Informatique Fondamentale de l'Université d'Orléans, BP 6759-45067, Orléans Cedex 2, France, 1992.

[Liu96] Y. A. Liu, *Incremental Computation: A Semantics-Based Systematic Transformational Approach*, Ph.D. Dissertation, Computer Science Department, Cornell University, January 1996.

[MTH90] R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML*, MIT Press, 1990.

[MTH97] R. Milner, M. Tofte, R. Harper, and D. MacQueen. *The Definition of Standard ML (Revised)*, MIT Press, 1997.

[Muc97] S. S. Muchnick. *Advanced Compiler Design Implementation*, Morgan Kaufmann Publishers, 1997.

[Myc81] A. Mycroft. *Abstract Interpretation and Optimising Transformations for Applicative Programs*, Ph.D. thesis, University of Edinburgh, 1981.

[NN92] F. Nielson, and H. R. Nielson. *Two-level Functional Languages*, Cambridge University Press, 1992.

[Nik90] R. S. Nikhil. "The Semantics of Update in a Functional Database Pro-

gramming Language," *Advances in Database Programming Languages*, eds. F. Bancilhon and P. Buneman, pp.403-421, ACM Press, 1990.

[Ode91] M. Odersky. "How to Make Destructive Update Less Destructive," *Proceedings on* 18^{th} *ACM Symposium on Principles of Programming Languages*, 1991.

[Pai81] R. Paige. *Formal Differentiation: A Program Synthesis Technique*, UMI Research Press, 1981.

[PK82] R. Paige, and S. Koenig. "Finite Differencing of Computable Expressions," *ACM Transactions on Programming Languages and Systems*, pp. 402-454, Vol. 4, No. 3, July 1982.

[Pai86] R. Paige. "Programming with Invariants," *IEEE Software*, pp. 560-69, Vol. 3, No. 1, January 1986.

[Pai89] R. Paige. "Real-Time Simulation of A Set Machine on a RAM," *Proceedings on ICCI 89*, May 1989, also in *Computing and Information*, Vol. II, eds. R. Janicki and W. Koczkodaj, pp. 69-73, Canadian Scholars Press, 1989.

[Pai90] R. Paige. "Symbolic finite differencing - Part I.", *Third European Symposium on Programming, ESOP '90*, pp. 36-56, Edited by N. D. Jones, *LNCS* 432, Springer-Verlag, 1990.

[Pai94] R. Paige. "Viewing a Program Transformation System at Work," Joint 6th International Conference on Programming Language Implementation and Logic programming (PLILP) and 4th International Conference on Algebraic and Logic Programming (ALP), *LNCS 844*, eds. M. Hermenegildo and J. Penjam, Springer-Verlag, September 1994, pp. 5-24.

[Pai97] R. Paige. "Future Directions in Program Transformations," *ACM SIGPLAN Notices*, pp. 94-98, Vol. 32, No. 1, January 1997.

[Pip97] N. Pippenger. "Pure versus Impure Lisp," *Transactions on Programming Languages ans Systems*, Vol. 19, No. 2, pp. 223-238, March 1997.

[Rob65] J. A, Robinson. "A Machine-Oriented Logic Based on the Resolution Principle," *Journal of the ACM*, pp. 23-41, Vol. 8, No. 12, December 1965.

[Scm85] D. A. Schmidt. "Detecting Global Variables in Denotational Specifications," *ACM Transactions on Programming Languages and Systems*, Vol. 7, No. 2, pp. 299-310, 1985.

[Scm88] D. A. Schmidt. "Detecting Stack-Based Environments in Denotational Definitions," *Science of Computer Programming*, Vol. 11, pp 107-131, 1988.

[Scw75a] J. T. Schwartz. "Optimization of Very High Level Languages – I," *Journal of Computer Languages*, Vol. 1, No. 2, pp. 161-194, June 1975.

[Scw75b] J. T. Schwartz. "Optimization of Very High Level Languages – II," *Journal of Computer Languages*, Vol. 1, No. 3, pp. 197-218, September 1975.

[SDD86] J. T. Schwartz, R. B. K. Dewar, E. Dubinsky, E. Schonberg. *Programming With Sets: An Introduction to SETL*, Springer-Verlag Press, 1986.

[Ses88] P. Sestoft. *Replacing Function Parameters By Global Variables*, Master's thesis, DIKU, University of Copenhagen, October 1988.

[Ses89] P. Sestoft. "Replacing Function Parameters By Global Variables," *Proceedings on Conference on Functional Programming Languages and Computer Architecture*, pp. 39-53, ACM Press, London, September 1989.

[Set89] R. Sethi. *Programming languages: Concepts and Constructs*, Addison-Wesley Press, 1989.

[SK95] K. Slonneger, and B. L. Kurtz. *Formal Syntax and Semantics of Programming Languages*, Addison-Wesley Press, 1995.

[Smi90] D. R. Smith. "KIDS: A Semi-Automatic Program Development Sys-

tem," *IEEE Transactions on Software Engineering*, Special Issue on Formal Methods, Sep. 1990.

[Sny90] W. K. Snyder. "The SETL2 Programming Language," *Technical Report 490*, Computer Science Department, Courant Institute of Mathematical Sciences, New York University, September 1990.

[Tar83] R. E, Tarjan. *Data Structures and Network Algorithms*, SIAM Press, 1983.

[TWM95] D. N. Turner, P. Wadler, and C. Mossin. "Once Upon A Type," *Proceedings on 7^{th} International Conference on Functional Programming and Computer Architecture*, June 1995.

[WC98] M. Wand, and W. D. Clinger. "Set Constraints for Destructive Array Update Optimization," *Proceedings of ICCL'98*, May 1998.

[WM95] R. Wilhelm, and D. Maurer. *Compiler Design*, English Edition, Addison-Wesley Press, 1995.

[Win93] G. Winskel. *The Formal Semantics of Programming Languages*, MIT Press, 1993.

[WF91] A. K. Wright, and M. Felleisen. "A Syntactic Approach to Type Sound-

ness," *Rice Technical Report TR91-160*, Computer Science Department, Rice University, revised June 1992.

[Yun97] C. Yung. "Extending Typed Lambda Calculus to Sets," *The Proceedings of MASPLAS'97, in cooperation with ACM SIGPLAN,* Stroudsburg PA, April 1997.

[Yun98] C. Yung. "EAS: An Experimental Applicative Language With Sets," *Proceedings of 1998 Mid-Atlantic Student Workshop on Programming Languages and Systems (in cooperation with ACM SIGPLAN)*, New Brunswick NJ, April 1998.