DYNAMIC SLICING IN HIGHER-ORDER PROGRAMMING LANGUAGES

A Dissertation

by

SANDIP K. BISWAS

Department of CIS

University of Pennsylvania

Philadelphia, PA 19104

`sbiswas@saul.cis.upenn.edu`

Dissertation Supervisor:

Carl A. Gunter, University of Pennsylvania

Dissertation Committee:

John Field, IBM T.J. Watson Research Center

Insup Lee, University of Pennsylvania

Dale Miller, University of Pennsylvania

Scott Nettles, University of Pennsylvania

Contents

List of Tables

Abstract

Dynamic slicing is a technique for isolating segments of a program that (potentially) contribute to the value computed at a point of interest. Dynamic slicing for restricted first-order imperative languages has been extensively studied in literature. In contrast, little research has been done with regard to the slicing of higher-order programs. Most first-order imperative programming languages are statement-based while most higher-order programming languages are expression-based. Unlike first-order programs, higher-order programs have no simple concept of static control flow: control flow depends on the binding of formal parameters to actuals. Because of these differences, formalising a definition of slicing for higher-order programs involves some novel concepts.

The aim of the work, presented here, is to extract 'executable' slices of higher-order programs solely from the execution trace. In the absence of assignments, i.e. in purely functional programs, dynamic slices satisfying very strong criteria can be extracted. This is because purely functional languages have a demand-driven evaluation strategy. A realistic higher-order programming language, like Standard ML (SML), uses imperative features like assignments and exceptions. We provide algorithms to compute dynamic slices of programs containing such features.

It is shown that, just like first-order programs, higher-order programs can be instrumented to collect data, regarding its dynamic slice, during execution. We have implemented a tool which performs such instrumentation on core SML programs. Experiments conducted, with the tool, throw light on the utility and limitations of dynamic slicing as technique for analysing higher-order programs.

Chapter 1

Introduction

A *program slice* consists of the parts of a program that (potentially) affect the values computed at some point of interest, referred to as a *slicing criterion*. Weiser, in his seminal work [65, 67][1], was the first to give formal definitions and algorithms in this area. In [66], he presents a justified claim that programmers mentally compute the slice associated with a break-point, when debugging a program. A first-order imperative program, without procedure calls, is essentially a sequence of statements. Slicing, for such programs, involves isolating a set of statements to be included in the *slice*. The new program, obtained after the deletion of the subset of statements to be excluded, is still an executable program and is a program slice, if it displays the same *behavior* at the point of interest. This is highly useful in debugging: if we are getting a wrong value at a break-point, then, during re-execution, we would like to execute only those statements which contributed to the value computed at the breakpoint. Such an approach can significantly reduce the time required to debug a program.

The informal definition of program slicing talks about isolating parts of a program that potentially affect the values computed at some point of interest. This isolation, of parts of a program, may be for a specific input to the program, in which case it is called *dynamic slicing*. On the other hand, we may want to isolate parts of a program which include all parts of a program that potentially affect the values computed at some point of interest, over all possible inputs, in which case it is called *static slicing*.

---

[1]One of the early presentations [67], contains seriously flawed definitions. These have been pointed out and corrected in [44].

For example, in Fig 1.1, the program (a) has the program (b) as its *executable dynamic slice*, if the slicing criterion is the value of x printed out at the end of program (a). The statement "x++" is not included in the slice because the incremented value of x produced at the end of the loop is discarded.

```
x = 1 ;                                          z = 7 ;
y = 2 ;                                          while (z > 0) do
z = 7 ;                                            if (z == 6) then y = 3 ;
while (z > 0) do                                   z − − ;
  if (z == 6) then y = 3 ;                       end ;
  x++ ;                                          if (y == 3) then x = 9 else  ;
  z − − ;                                        print(x) ;
end ;
if (x > 9) then y = 4 ;
if (y == 3) then x = 9 else x = x + 10 ;
print(x) ;
```

(a) $\hspace{6cm}$ (b)

Figure 1.1

In Fig 1.1(a), if the initial value of z is set to 6 instead of 7 then the statement "z-- ; " does not explicitly 'contribute' to the value of x printed out at the end of the program: it merely ensures that the while loop terminates. There are approaches to dynamic slicing that do not include such statements, e.g [6]. Such slices, of course, are not executable programs. Agrawal et al,[4, 5], have developed a sophisticated debugging tool, SPYDER, for a small subset of the programming language C, based on this principle.

The transition from intuition to a well-formed definition, for executable slices of higher-order programs, is not so obvious a step. This can be seen from the following example:

$$(\, (\lambda y.\, y)(\lambda x.\, 4) \,)(20 + 30)$$

Since the term $(20 + 30)$ does not raise an arithmetic exception, it does not 'contribute' to the value returned by the original program. Unlike the first-order case, simple deletion of the term $(20 + 30)$ does not leave behind a program that returns the same answer.

2

Replacement of the subterm $(20 + 30)$ by some constant/canonical integer value works for this example. But if a higher-order term does not contribute to the answer then a replacement strategy may not be obvious.

```
let fun F  f x y = let val z = f x
                   in
                   if (y=1) then 90 else z
                   end
in
 F (G f1) 2 3  + F (H f2) 0 1
end
```

Figure 1.2

A higher-order program, written in SML, is presented in Fig 1.2. The variables `G`, `H, f1` and `f2` in the program are left undefined as they are not relevant to the point we want to illustrate. The subterm `(H f2)` does not contribute to the value returned by the program as the term `(F (H f2) 0 1)` evaluates to the constant `90`. To define an executable dynamic slice for this program, we need to present an altered version of the program in which the term `(H f2)` is either absent or not evaluated. The alteration is definitely not as trivial as term deletion or replacement by some canonical term, as `(H f2)` evaluates to a function of type (int $->$ int).

Another concept which we would like to present informally is that of *relevant slicing*. This concept was developed by Agrawal et al [7] in the context of incremental regression testing. Let us consider a software package that has been developed and subsequently tested by a large and comprehensive test suite. If the next version of the software package made only small and scattered changes to its predecessor then we would like to know whether a particular test, in the suite, need not be rerun because the changes made to the software were not *relevant* to the execution of the test. On a more formal footing, let us consider the language **L** presented in Page 8. Let the set of allowable changes to a program be changes made to expressions e within statements.

The *relevant slice* of a program, with respect to a slicing criterion, not only includes the dynamic slice, with respect to the slicing criterion, but also statements that were executed

3

and did not influence the slicing criterion but could have influenced it, had they evaluated differently.

Consider the programs presented in Fig 1.3.

```
a = 2 ;
b = 2 ;                              b = 2 ;
if (a > 3) then b = b * 8;        if (b < 10)  then c = 9 ;
if (b < 10)  then c = 9 ;                      else ;
             else c = 11 ;       print(c) ;
print(c) ;
            (a)                                    (b)
```

```
            a = 2 ;
            b = 2 ;
            if (a > 3) then ;
            if (b < 10)  then c = 9 ;
                         else ;
            print(c) ;
                        (c)
```

Figure 1.3

The program (b) represents the dynamic slice of the program (a), for a slicing criterion represented by the output of the program. The program (c) is the relevant slice of (a), with respect to the same slicing criterion: it is the complete execution slice of (a). The statement "a = 2 ;" needs to be included in the relevant slice because if it were changed to "a = 4 ;" the output of the program would change to 11. For similar reasons, the conditional "if (a > 3) then ; " must be included in the relevant slice.

The computation of relevant slices involves the defining and computing of *potential dependencies* [7]. This computation involves more than a traversal of the execution trace of a program: a static data-flow analysis for the collecting reaching definitions is required.

1.1 An Outline of Research Results

The informal introduction, above, attempted to provide a feel for the fact that dynamic slicing for statement-based first-order languages is a well-understood concept with formal definitions, algorithms and correctness proofs. But this is not the case for an expression-based higher-order language whose operational semantics is presented as a natural semantics [54, 37]. This thesis presents a formal framework in which a dynamic slice of a higher-order program can be defined. This was a non-issue for a first-order language, since the deletion of an arbitrary set of statements leaves behind a syntactically correct program.

We go on to present an algorithm for the computation of a dynamic slice for purely functional programs. The algorithm for the computation of the dynamic slice is presented as a natural semantics and its correctness is proved. We provide algorithm-independent characterisations of the dynamic slice we compute. Once we have an extensional characterisation of dynamic slices a natural question to be asked is, can we compute a static approximation to it? We provide an answer in the affirmative through the use of a set-based analysis technique. We discuss how this static analysis may find use in the isolation of dead-code in higher-order programs.

We next present an algorithm for slicing in a higher-order language, with imperative features: assignments and exceptions. As shown in [40], the computation of executable dynamic slices for first-order imperative languages requires a closure operation, over a set of relations defined on the execution trace, because of multiple executions of a statement in a while loop. If the operational semantics of a higher-order imperative programming language is presented as a natural semantics then the execution trace of a terminating computation is a proof-tree. This is very different from the case for for a statement-based first-order language where the execution trace is a sequence of statements. Multiple executions of a term occurs because of multiple call-sites of a closure. The closure operation to be performed in this case is much more subtle, particularly in the presence of exceptions. The relation, over which the closure is to be performed, is deeply rooted in semantics of control-flow in the presence of exceptions and assignments.

Apart from simpler correctness proofs, the presentation of the algorithm for the computation of dynamic slices, as a natural semantics, has an immediate benefit: a term can be instrumented to collect data regarding its dynamic slice. We define a formal translation of a program P into an annotated program P' and prove that the execution of P' correctly computes the dynamic slice of the execution P. We have implemented a translator for the whole of core SML and perform experiments to investigate the utility and limitations of slicing techniques for analysing higher-order languages.

Chapter 2

A Formal Presentation of Slicing for First-Order Imperative Programs

The previous chapter attempted to provide an informal and intuitive understanding of the concepts involved in program slicing, for first-order programs, and their possible applications. This chapter presents a formal definition of the concepts involved. Since program slicing can be viewed as a source-to-source transformation on programs, a formal definition of slicing must be based on the semantics of the programming language. The semantics of a programming language may be denotational [58] or operational [25]. The formal definitions presented in Section 2.1 are based on denotational semantics. A denotational framework was chosen as it allows us to define a broad set of definitions very succinctly, independent of the algorithms used to compute the slices, and independent of the presentation style of the operational semantics.

The definitions and algorithms presented in this chapter are essentially a review of previous research in this area. They are not meant to be comprehensive, but are meant to show that concepts, developed subsequently, for higher-order programs are a natural extension of the concepts formulated for first-order languages. A very comprehensive survey of the research in program slicing may be found in [62].

The formulation presented in Section 2.1 is from Venkatesh [63]. The programming

language **L**, on which analysis is carried out in the following sections, is given by the following grammar:

$$s \quad ::= \quad i := l : e$$
$$| \quad \text{if } l : e \text{ then } s_1 \text{ else } s_2$$
$$| \quad \text{while } l : e \text{ do } s$$
$$| \quad s_1 \,;\, s_2$$
$$|$$

L is a statement-based language: a program in **L** is a sequence of statements. The non-terminal e denotes expressions, whose syntax is left unspecified. All expressions in this language are assigned unique labels: subexpressions are not individually labelled. The standard semantics of the language is presented in Table 2.1.

The semantic function **E** and the semantic domain *value* are left unspecified. It is assumed that expressions consist of constant time operations over variables and constants, and that they have no side effects. The language is given a strict semantics, i.e. the update function associated with the store is assumed to be strict in all three of its arguments.

Domains:
$\sigma \in store = id \rightarrow value$

Functions:
$\mathbf{E} : \quad exp \rightarrow store \rightarrow value$
$\mathbf{S} : \quad stmt \rightarrow store \rightarrow store$

$\mathbf{S}[\![i := l : e]\!] \;=\; \lambda\sigma.\; \sigma[(\mathbf{E}[\![e]\!]\sigma)/i]$
$\mathbf{S}[\![\text{if } l : e \text{ then } s_1 \text{ else } s_2]\!] = \lambda\sigma.\; \mathbf{E}[\![e]\!]\sigma \;\rightarrow\; \mathbf{S}[\![s_1]\!]\sigma \,,\, \mathbf{S}[\![s_2]\!]\sigma$
$\mathbf{S}[\![\text{while } l : e \text{ do } s]\!] = \text{fix}(\lambda f.\lambda\sigma.\; \mathbf{E}[\![e]\!]\sigma \;\rightarrow\; f(\mathbf{S}[\![s]\!]\sigma) \,,\, \sigma\,)$
$\mathbf{S}[\![s_1 \,;\, s_2]\!] \;=\; \lambda\sigma.\; \mathbf{S}[\![s_2]\!](\mathbf{S}[\![s_1]\!]\sigma)$

Table 2.1 : Standard Denotational Semantics of **L**

Notation: $\Lambda(s)$ denotes the set of all labels in an **L**-program s.

As **L** is a statement-based language, given any $L \subseteq \Lambda(s)$, it is possible to inductively construct a syntactically correct program that includes only those statements, whose labels are included in the set L, and its *control dependencies*. For a block-structured language like **L**, if a statement s is immediately contained in a block, defined by a control construct like an if-then-else statement or a while-loop, then s is said to be control-dependent on the control-construct. Such an inductive construction is given, in Table 2.2, by the function $Syn(s\,,\,L)$.

$Syn(s\,,\,L) =$
    case s of
          $[\![\, i := l : e \,]\!]$: if $l \in L$ then s else ε
          $[\![\, \texttt{if } l : e \texttt{ then } s_1 \texttt{ else } s_2 \,]\!]$:
              if $Syn(s_1\,,\,L) = Syn(s_2\,,\,L) = \varepsilon$ and $(l \notin L)$
              then ε
              else $[\![\, \texttt{if } l : e \texttt{ then } Syn(s_1\,,\,L) \texttt{ else } Syn(s_2\,,\,L) \,]\!]$
          $[\![\, \texttt{while } l : e \texttt{ do } s' \,]\!]$:
              if $Syn(s'\,,\,L) = \varepsilon$ and $(l \notin L)$
              then ε
              else $[\![\, \texttt{while } l : e \texttt{ do } Syn(s'\,,\,L) \,]\!]$
          $[\![\, s_1\,;s_2 \,]\!]$: $[\![\, Syn(s_1\,,\,L); Syn(s_2\,,\,L) \,]\!]$

Table 2.2 : $Syn(s\,,\,L)$

2.1   A Denotational Formulation of Slicing

In his formulation, Venkatesh uses a single parameter slicing criterion: the value of a specific variable at the end of a computation.

Definition: For any **L**-program s, a variable v used in s, a set of labels $L \subseteq \Lambda(s)$ and any initial store σ_0, the **L**-program $Syn(s\,,\,L)$ is called

- A **dynamic slice with respect to** $v\,,\,\sigma_0$ iff $(\mathbf{S}[\![\, s \,]\!]\sigma_0)(v) = (\mathbf{S}[\![\, Syn(s\,,\,L) \,]\!]\sigma_0)(v)$

- A **static slice with respect to** v iff $(\mathbf{S}[\![\, s \,]\!]\sigma_0)(v) = (\mathbf{S}[\![\, Syn(s\,,\,L) \,]\!]\sigma_0)(v)$ for all $\sigma_0 \in store$.

The above definitions of slices deal with syntactically correct programs: they cannot accommodate slices which are not executable programs. Hence, Venkatesh presents a denotational formulation of the intuitive concept of a statement influencing the value of a variable, at the end of a computation. He defines *contamination* of an expression and a semantics for propagation of contamination. A statement influences the value of a variable, at the end of the computation, if its contamination results in the value of the variable being contaminated at the end of the computation.

To denote contaminated computations, every value becomes a tagged value, *t_value*, tagged with a boolean flag with *true* indicating contamination. Hence, the store becomes a tagged store, *t_store* $\equiv$ *id* $\rightarrow$ *t_value*. The function $\mathbf{E}_{\text{instr}}$, in Table 2.3, takes in as input contaminated expressions, *c_exp* $\equiv l : (t, e)$, where the tag t indicates whether the expression labelled l has been marked contaminated. A contaminated statement, *c_stmt*, is similar to a *stmt*, except that expressions in the statement have now been replaced contaminated expressions, *c_exp*. The function $\mathbf{E}_{\text{instr}}$ is derived from the function $\mathbf{E}$ by associating a boolean tag, with the output value, that is a disjunction of the tags associated with all the values used by the expression, the tag associated with expression itself and the additional boolean parameter passed in as argument. The boolean parameter taken in by $\mathbf{E}_{\text{instr}}$ and $\mathbf{S}_{\text{instr}}$ essentially indicates whether the value, on which a given statement/expression is control dependent, is contaminated. A complete instrumented semantics is given in Table 2.3.

Given a statement s, $Cont(s, l)$ is a statement in *c_stmt*, in which every expression, $(l' : e)$ where $l' \not\equiv l$, is replaced by $(l' : (false, e))$ and the expression $(l : e)$ is replaced by $(l : (true, e))$.

Definition: Let s be an **L**-program, v a variable in s and σ_0 be an initial store. Let $L \subseteq \Lambda(s)$ be the set of labels defined by,

$$L = \{ \, l \mid ([\mathbf{S}_{\text{instr}}[\![\, Cont(s, l) \,]\!] \, false \, \lambda i. \, (\sigma_0(i), \, false)] \, (v)) \downarrow 2 = true \, \}$$

The set L is defined to be the **dynamic backward closure slice**, DBC(s, v, σ_0), of s with respect to the variable v and the initial store σ_0.

A DBC slice includes exactly those statements whose contamination contaminates the slicing criterion. A DBC slice is what is computed by the algorithm presented by

Domains:

$(\nu, \beta) \in t\_value = value * bool$

$\tau \in t\_store = id \rightarrow t\_value$

Functions:

$\mathrm{E}_{\mathrm{instr}} : c\_exp \rightarrow bool \rightarrow t\_store \rightarrow t\_value$

$\mathrm{S}_{\mathrm{instr}} : c\_stmt \rightarrow bool \rightarrow t\_store \rightarrow t\_store$

$\mathrm{S}_{\mathrm{instr}}[\![\, i := l : p \,]\!] = \lambda\beta\tau.\ \tau[(\mathbf{E}_{\mathrm{instr}}[\![\, p \,]\!]\beta\tau)/i]$

$\mathbf{S}_{\mathrm{instr}}[\![\, \texttt{if}\ l : p\ \texttt{then}\ s_1\ \texttt{else}\ s_2 \,]\!] =$
$\qquad \lambda\beta\tau.\ \ \text{let}\ (\nu, \beta_1) = \mathbf{E}_{\mathrm{instr}}[\![\, p \,]\!]\beta\tau$
$\qquad\qquad \text{in}\ \ \nu \rightarrow \mathbf{S}_{\mathrm{instr}}[\![\, s_1 \,]\!]\beta_1\tau\ ,\ \mathbf{S}_{\mathrm{instr}}[\![\, s_2 \,]\!]\beta_1\tau$

$\mathbf{S}_{\mathrm{instr}}[\![\, \texttt{while}\ l : p\ \texttt{do}\ s \,]\!] =$
$\qquad \text{fix}(\lambda f.\lambda\beta\tau.\ \ \text{let}\ (\nu, \beta_1) = \mathbf{E}_{\mathrm{instr}}[\![\, p \,]\!]\beta\tau$
$\qquad\qquad \text{in}\ \ \nu \rightarrow f\,\beta_1\,(\mathbf{S}_{\mathrm{instr}}[\![\, s \,]\!]\beta_1\tau)\ ,\ \tau\,)$

$\mathbf{S}_{\mathrm{instr}}[\![\, s_1\,;\,s_2 \,]\!] = \lambda\beta\tau.\ \mathbf{S}_{\mathrm{instr}}[\![\, s_2 \,]\!]\beta(\mathbf{S}_{\mathrm{instr}}[\![\, s_1 \,]\!]\beta\tau)$

Table 2.3 : Instrumented Semantics of **L**

Agrawal and Horgan in [6]. The important point to be observed is that the execution of $Syn(s\,,\,\mathrm{DBC}(s\,,\,v\,,\,\sigma_0))$ need not return the same answer for the slicing criterion v. In fact, the execution of $Syn(s\,,\,\mathrm{DBC}(s\,,\,v\,,\,\sigma_0))$ need not even terminate. For example, in Fig 1.1, if the initial value of z in program (a) were set to 6 then contaminating ``z-- ; '' would not contaminate the value of $x$. This is because the value of $y$, which sets the value of $x$, is set in the first pass through the loop. Since ``z-- ; '' is not included in the DBC, its execution no longer terminates.

A statement gets included in a DBC slice because along one specific control flow path its contamination contaminates the slicing criterion. But, for a given statement s_1, included in a DBC slice, no attempt is made to include all statements, in the program, whose contamination contaminates s_1. This is the reason $\mathrm{DBC}(s\,,\,v\,,\,\sigma_0)$ may fail to be equivalent to the original program s, with respect to a variable v and a store σ_0. Hence, for transforming a DBC slice into an executable dynamic slice we need a closure technique. This technique uses the augmented instrumented semantics defined in Table 2.4. All semantic

functions, used in Table 2.4, are assumed to be $-$ preserving. Every defining clause for the semantic function S$'_{\text{instr}}$ is assumed to have a guard which checks that the argument β does not equal $-$. If it does, it immediately returns $-$. The semantic function S'_{instr}

Domains:

$(\nu\,,\,\beta)\ \in\ t\_value = value * bool$

$\tau\ \in\ t\_store\ =\ id \to t\_value$

Functions:

$\text{S}'_{\text{instr}}\ :\ \ c\_stmt\ \to\ \mathcal{P}(label)\ \to\ bool\ \to\ t\_store\_\ \to\ t\_store\_$

$\text{S}'_{\text{instr}}[\![\,i := l : p\,]\!]\ =\ \lambda L\beta\tau.\ \ \text{let}\ (\nu\,,\,\beta_1) = \mathbf{E}_{\text{instr}}[\![\,p\,]\!]\beta\tau$
$$\text{in}\ (l \in L \,\text{and}\, \beta_1 = true)\ \to\ -\ ,\ \tau[(\nu\,,\,\beta_1)/i]$$

$\mathbf{S}'_{\text{instr}}[\![\,\texttt{if}\ l : p\ \texttt{then}\ s_1\ \texttt{else}\ s_2\,]\!] =$
$\lambda L\beta\tau.\ \text{let}\ (\nu\,,\,\beta_1) = \mathbf{E}_{\text{instr}}[\![\,p\,]\!]\beta\tau$
$$\text{in}\ (l \in L \,\text{and}\, \beta_1 = true)\ \to\ -\ ,\ (\nu\ \to\ \mathbf{S}'_{\text{instr}}[\![\,s_1\,]\!]L\beta_1\tau\ ,\ \mathbf{S}'_{\text{instr}}[\![\,s_2\,]\!]L\beta_1\tau\,)$$

$\mathbf{S}'_{\text{instr}}[\![\,\texttt{while}\ l : p\ \texttt{do}\ s\,]\!] =$
$\ \text{fix}(\lambda f.\lambda L\beta\tau.\ \ \text{let}\ (\nu\,,\,\beta_1) = \mathbf{E}_{\text{instr}}[\![\,p\,]\!]\beta\tau$
$$\text{in}\ (l \in L \,\text{and}\, \beta_1 = true)\ \to\ -\ ,\ (\nu\ \to\ f(\mathbf{S}'_{\text{instr}}[\![\,s\,]\!]\beta_1\tau)\ ,\ \tau\,)$$

$\mathbf{S}'_{\text{instr}}[\![\,s_1\,;\,s_2\,]\!]\ =\ \lambda L\beta\tau.\ \mathbf{S}'_{\text{instr}}[\![\,s_2\,]\!]\,L\,\beta\,(\mathbf{S}'_{\text{instr}}[\![\,s_1\,]\!]L\beta\tau)$

Table 2.4 : Augmented Instrumented Semantics of **L**

---

takes in a set of labels L and returns $-$, if during the computation of its argument, a statement included in L is executed and returns a contaminated value. Using the semantic function S'_{instr} we can isolate the set of statements whose contamination leads to any statement in DBC($s\,,\,v\,,\,\sigma_0$) to become contaminated. This set of statements can be used as an argument to S'_{instr} to obtain a possibly larger set of statements. The iteration can be continued till we reach a limit. The limiting set of statements is an executable dynamic slice.

Definition: Let s be a program, v a variable used in s and σ_0 any initial store. Let L be the limit of the sequence $L_0\,,\,L_1\,,\,\ldots$ where

$L_0 = \text{DBC}(s\,,\,v\,,\,\sigma_0)$

$L_{i+1} = \{\,l\ |\ S'_{\text{instr}}[\![\ Cont(s\,,\,l)\,]\!]\ L_i\ false\ (\lambda i.\,(\sigma_0(i)\,,\,false)) = -\,\}$

The set L is called a **dynamic backward executable slice**, DBE(s, v, σ_0), of s with respect to a variable v and the initial store σ_0.

Theorem 2.1.1 *For any program s, a variable v used in s and an initial store σ_0, if $s' \equiv Syn(s, DBE(s, v, \sigma_0))$ then $(\mathbf{S}[\![s]\!]\sigma_0)(v) = (\mathbf{S}[\![s']\!]\sigma_0)(v)$, i.e. DBE($s$, v, σ_0) is a dynamic slice with respect to v, σ_0.*

Definition: Let s be a program, v a variable used in s. Let $L \subseteq \Lambda(s)$ be a set of labels such that $L \supseteq \cup \, \mathrm{DBE}(s, v, \sigma_0)$ for all $\sigma_0 \in store$. Then L is defined to be a **static backward executable slice**(SBE) of s with respect to the variable v.

Theorem 2.1.2 *For any program s and a variable v used in s, if $s' \equiv Syn(s, SBE(s, v))$ then $(\mathbf{S}[\![s]\!]\sigma_0)(v) = (\mathbf{S}[\![s']\!]\sigma_0)(v)$ for any σ_0, i.e. SBE(s, v) is a static slice with respect to v.*

2.2 Algorithms for Program Slicing

The previous section presented a denotational definition of static and dynamic slices. An instrumented denotational semantics was supplied to characterise statements, which were to included in the DBE. The instrumented semantics did not, however, provide for a technique to compute the DBE of a program s with respect to v, σ_0. In this section, we will present the technique developed by Korel and Laski [40], for the computation of dynamic slices.

Given a program s, in the language **L**, and an initial memory σ_0,

- $\mathcal{T}$ denotes the execution trace of s. As s is a program whose top-level expressions are labelled, an execution trace can be defined as the sequence of labels of expressions that were executed.

- $\mathcal{T}_i$ denotes the ith label in the sequence $\mathcal{T}$.

- $\mathcal{T}|_i$ is a sequence obtained by restricting $\mathcal{T}$ to its first i elements.

- $\Pi_L(\mathcal{T})$, where L is a set of labels, denotes the sequence of labels obtained by restricting $\mathcal{T}$ to labels from L.

- $Use(l)$ denotes the set of variables in the expression labelled l.

- If an expression e labelled l is a part of an assignment statement $x := l : e$ then $Def(l) \equiv x$.

- The *Data-Data(DD) Relation*, on an execution trace $\mathcal{T}$, is a subset of $N * N$, such that $i\,DD\,j$ iff $i < j$ and there exists a variable v such that $v \equiv Def(\mathcal{T}_i)$ and $v \in Use(\mathcal{T}_j)$ and for any k, $i < k < j$, $v \not\equiv Def(\mathcal{T}_k)$.

- The *Identity(IR) Relation*, on an execution trace $\mathcal{T}$, is a subset of $N * N$, such that $i\,IR\,j$ iff $\mathcal{T}_i \equiv \mathcal{T}_j$.

- The *Test-Control(TC) Relation*, on an execution $\mathcal{T}$, is a subset of $N * N$.

  For a statement `if` $l : p$ `then` s_1 `else` s_2 the labels in $\Lambda(s_1)$ and $\Lambda(s_2)$ are defined to be in the scope of influence of the label l.

  For a statement `while` $l : p$ `do` s the labels in $\Lambda(s)$ are defined to be in the scope of influence of the label l.

  $i\,TC\,j$ iff $\mathcal{T}_j$ is in the scope of influence of $\mathcal{T}_i$ and for all k, $i < k < j$, $\mathcal{T}_k$ is the scope of influence of $\mathcal{T}_i$.

Definition: Let $\mathcal{T}$ be the execution trace of a program s, on input σ_0. A **slicing criterion** C is a tuple $\langle q, V \rangle$, where q is a position in the execution trace $\mathcal{T}$ and V is a subset of the variables in s.

This is a more refined slicing criterion than discussed in the previous section. It has two parameters instead of one: a position in the execution trace is asked for. Another way of looking at it is, the previous slicing criterion had the q parameter fixed to the position beyond the last label in the execution trace.

Definition: Given a slicing criterion $C \equiv \langle q, V \rangle$, a **dynamic slice** of s with respect to C, on input σ_0, is any executable program s' that is obtained from s by deletion of zero or more statements from it and when executed on σ_0, produces an execution trace $\mathcal{T}'$ for which there exists a position q' such that,

1. $\mathcal{T}'|_{q'} \equiv \Pi_{\Lambda(s')}(\mathcal{T}|_q)$

14

2. for all $v \in V$, the value of v before the execution of $\mathcal{T}_q$ exactly equals the value of v before the execution of $\mathcal{T}'_{q'}$.

3. $\mathcal{T}_q \equiv \mathcal{T}_{q'}$

Given the execution trace $\mathcal{T}$ of a program s, on input σ_0 and a slicing criterion $C \equiv \langle q, V \rangle$, the dynamic slice is computed by an iterative process.

$$S_0 \quad = \quad Last\_Def(q, V) \cup Last\_Control(q)$$

$$\text{where} \quad Last\_Def(q, V) = \{p \mid Def(\mathcal{T}_p) \equiv v \in V \text{ and for any } n, p < n < q, v \not\equiv Def(\mathcal{T}_n)\}$$

$$Last\_Control(q) = \{p \mid p \, TC \, q\}$$

$$S_{i+1} \quad = \quad S_i \cup \{p \mid p \, (DD + IR + TC) \, r \text{ where } r \in S_i\}$$

The above iteration converges to a limit S.

The dynamic slice of s with respect to C, on input σ_0, includes exactly the statements labelled $\{\mathcal{T}_p \mid p \in S\}$.

A counterpart of the algorithm, presented above, for static slicing would be an algorithm which works for all initial memories. The transition to a static slicing algorithm is very gradual and intuitive. It involves the following steps:

- The first component of a slicing criterion $C \equiv \langle q, V \rangle$, used in the above algorithm, is a position in the execution trace $\mathcal{T}$. For static slicing, there is no execution trace available, hence a slicing criterion is given by $C \equiv \langle l, V \rangle$ where l is the label associated with a statement.

- The DD-Relation, defined above, relates $Last\_Def(q, Use(\mathcal{T}_q))$ to the position q. Analogously, we define a *data-dependence* relation relating statements, labelled l and m, if l defines a variable v, $v \in Use(m)$ and there is a path from l to m, in the control-flow graph for s, which does not have a definition of v in it. For the language **L**, the data-dependence relation is statically computable.

- The inverse of the TC-Relation, defined above, is actually a function mapping a position, p in $\mathcal{T}$, to another position q. The corresponding mapping on labels, from $\mathcal{T}_p$ to $\mathcal{T}_q$ is invariant over all execution traces and defines the inverse of the *control-dependence* relation. For the block-structured language **L**, the

15

control-dependence relation is computable by making one pass over the program.

- By performing a transitive closure on the relation
  ($data\text{-}dependence$ + $control\text{-}dependence$), the static slice can be computed. A detailed
  account of static slicing may be found in [55].

Chapter 3

Related Work

As has been mentioned before, there has been very little research in the area of slicing of higher-order programs. There have been two prominent approaches.

- Field and Tip [22] have a very detailed study of the concept of slicing associated with left-linear term rewriting systems(TRS)[38]. As the semantics of a programming language can be provided as a TRS [1, 19, 20], the techniques developed, in their general study of slicing for TRS, can be applied to define and compute slices associated with the evaluation of a program[21].

- Another approach has been developed by Abadi et al [2] to analyze and cache dependencies involved in the evaluation of λ-terms.

The semantics of SML [48] is defined as a natural deduction system and none of the approaches mentioned above can handle natural deduction systems. Of course, it is possible to provide translations from one style of presentation of semantics to another [3, 27]. But, we would prefer a more direct approach in the computation of dynamic slices for SML-programs. One of the principal reasons for this is that the definition of slices is heavily dependent on the style of presentation of the operational semantics since it is an intensional property. Unless, there is a canonical extensional definition of slices, e.g. based on denotational semantics of labelled terms, we cannot really be sure whether the translation from one style, of presentation of operational semantics, to another results in loss of information with respect to slices. Besides, a SML programmer, used to thinking

about evaluation in a natural semantics, will have to start thinking about evaluation in a different style, e.g. a rewrite semantics, if he wants use a slice of the computation. The next two sections present a brief review of the above approaches.

3.1 Slicing in Term Rewriting Systems

First, we present a formal definition of term rewriting systems. Then, we define the fundamental concept developed by Field and Tip: *context-rewriting*. Through an example, dynamic slices are then shown to be *contexts* with a certain set of properties.

A *signature* Σ is a finite set of function symbols along with a map *arity* from this set of function symbols to the set of natural numbers, such that for any $f \in \Sigma$, $arity(f)$ stands for the number of arguments accepted by f.

A *path* is a sequence of positive integers that designates a subtree by encoding a walk from the tree's root. The empty path, $(\ )$, designates the root of a tree; the path $(i_1\ i_2\ \ldots\ i_m)$ designates the i_m^{th} subtree of the subtree indicated by the path $(i_1\ i_2\ \ldots\ i_{m-1})$. Roots of subtrees are numbered, starting from the left, beginning with 1. Paths are ordered by the relation, $\preceq$, which is the prefix relation. The operation '·' denotes the concatenation of paths.

A *tree* T is a set of paths such that (i) it possesses unique root, for all $t \in T$, $\mathrm{root}(T) \preceq t$ (ii) For all p, q, r such that $p \preceq q \preceq r$, if $p, r \in T$ then $q \in T$.

Definition: Let Σ be a signature, $\mathcal{V}$ be a set of variables, and T be a tree. Let μ be a total mapping from T to $(\Sigma \cup \mathcal{V})$ and p be a path. Then a pair $\langle p\, ,\, \mu \rangle$ defines a **context** iff:

(i) For all $t \in T$ and $s \in \Sigma \cup \mathcal{V}$ such that $\mu(t) = s$, if $t \cdot i \in T$ then $i \leq \mathrm{arity}(s)$.

(ii) If $T \neq \emptyset$ then $p = \mathrm{root}(T)$.

Given a context $C \equiv \langle p\, ,\, \mu \rangle$, $\mathcal{O}(C)$ denotes the domain of μ.

Definition: A context C is a **subcontext** of D, $C \sqsubseteq D$, iff all paths common to both their domains are mapped to the same symbol and one of the following holds: (i) C and

18

D are non-empty and $\mathcal{O}(C) \subseteq \mathcal{O}(D)$. (ii) C and D are empty and $C \equiv D$ (iii) C is empty, D is not and $\text{root}(C) = q \cdot i \in \mathcal{O}(D)$, and $q \in \mathcal{O}(D)$.

A path corresponding to a missing child, in a context C, is referred to as a *hole occurrence*.

A context C is a *term*, if it has no hole occurrences and $\text{root}(C) = ()$.

For any context C and a path p, $p \leftarrow C$ denotes an isomorphic context rooted at p obtained by rerooting C.

Two contexts C and D are isomorphic, $C \doteq D$, if $(() \leftarrow C) \equiv (() \leftarrow D)$.

The function *vars* takes in a term as an argument and returns the set of variables in the term as the result.

$C[D]$ denotes a context that is obtained from C by replacing the subcontext rooted at $\text{root}(D)$ by D.

Definition: A **term rewriting system** $\mathcal{R}$ over a signature Σ is any set of pairs $\langle L, R \rangle$ such that L and R are terms over Σ, L does not consist of a sole variable, and $vars(R) \subseteq vars(L)$.

A *substitution* σ is a finite function from the set of variables to the set of terms.

An $\mathcal{R}$-contraction $\mathcal{A}$ is a triple $\langle p, \alpha, \sigma \rangle$, where p is a path, α is rule of $\mathcal{R}$ and σ is a substitution.

Definition: A term T_0 **rewrites** to T_1 through a $\mathcal{R}$-contraction $\mathcal{A} \equiv \langle p, \alpha \equiv \langle L, R \rangle, \sigma \rangle$, $T_0 \xrightarrow{\mathcal{A}} T_1$, if $T_0 \equiv T_0[\sigma(p \leftarrow L)]$ and $T_1 \equiv T_0[\sigma(p \leftarrow R)]$.

A *reduction* $\rho : T_0 \longrightarrow^* T_n$ is a sequence of contractions $\mathcal{A}_1 \mathcal{A}_2 \ldots \mathcal{A}_n$ where:

$$T_0 \xrightarrow{\mathcal{A}_1} T_1 \xrightarrow{\mathcal{A}_2} T_2 \ldots T_{n-1} \xrightarrow{\mathcal{A}_n} T_n$$

A context C rewrites to a context C', $C \longrightarrow^* C'$, if the term T, obtained by instantiating every hole occurrence in C with a completely new variable, rewrites to T' such that T' is obtained from C' by some variable instantiation of the hole occurrences in C'.

Definition: Given a reduction $\rho : T \longrightarrow^* T'$, a **slicing criterion** associated with the reduction is any subcontext C' of the term T'.

What follows is an informal definition of program slices, to be illustrated by an example.

Definition: Let $\rho : T \longrightarrow^* T'$ be a reduction. A **slice** with respect to a slicing criterion C', is a subcontext C of T with the property that there exists a reduction ρ' such that $\rho' : C \longrightarrow^* D'$ for some $D' \sqsupseteq E'$, $E' \doteq C'$ and the reduction sequence $\langle C, \rho', D' \rangle$ is a *projection* of the original reduction ρ.

Consider the following term rewriting system:

R_1. $F(x, G\,y) \rightarrow H(x, x)$

R_2. $H(x, I(J\,y, z)) \rightarrow K(x, y)$

R_3. $K(I(x, L\,y), z) \rightarrow y$

R_4. $L(x) \rightarrow M(x)$

Consider the reduction ρ of the term $F(I(J(Kx), L(Ky)), G(L\,z))$.

$$F(I(J(K\,x), L(K\,y)), G(L\,z))$$
$$\xrightarrow{R_4} F(I(J(K\,x), L(K\,y)), G(M\,z))$$
$$\xrightarrow{R_1} H(I(J(K\,z), L(K\,y)), I(J(K\,z), L(K\,y)))$$
$$\xrightarrow{R_2} K(I(J(K\,z), L(K\,y)), L(K\,y))$$
$$\xrightarrow{R_4} K(I(J(K\,z), L(K\,y)), M(K\,y))$$
$$\xrightarrow{R_3} K\,y$$

Given such a reduction and a slicing criterion $(K\,y)$ the minimal slice is $F(I(J\bullet, L(K\,y)), G\bullet)$. The reduction ρ' associated with the slice is:

$$F(I(J\bullet, L(K\,y)), G\bullet)$$
$$\xrightarrow{R_1} H(I(J\bullet, L(K\,y)), I(J\bullet, L(K\,y)))$$
$$\xrightarrow{R_2} K(I(J\bullet, L(K\,y)), L(K\,y))$$
$$\xrightarrow{R_3} (K\,y)$$

The reduction sequence ρ' is a *projection* of the reduction ρ. Field and Tip display a sound technique for the computation of the minimal dynamic slice for left-linear term rewriting systems. The important point to note about their technique is:

- If the TRS is non-deterministic and we are using an interpreter with a specific strategy to pick up redexes then we cannot apply the same strategy to pick up redexes in the slice and execute the slice to obtain a context containing the slicing criterion. In fact, to execute the slice, we actually need to preserve information about the projection of the original reduction ρ.

This has both positive and negative aspects. On the negative side, we cannot use the same interpreter to execute the slice. This is because the use of the rewrite strategy of the standard interpreter may result in divergence of the slice being evaluated. To execute the slice, we need another interpreter which takes in a list, of redexes to be contracted, as a parameter. On the positive side, the slices may actually be much smaller. This is because the strategy used by the standard interpreter may choose to execute a specific redex whose final value is of no consequence to the answer. Only a termination of reduction on the redex may be relevant. For such a redex, we may need to preserve its entire set of dependencies, which may be large. The technique developed by Field and Tip avoids reducing such redexes.

3.2   Analysis and Caching of Dependencies

In [2], Abadi, Lampson and Lévy develop a concept similar to slicing for a completely different application: caching of the results of very expensive computations in purely functional programs. The application towards which their analysis is directed is a configuration management language called Vesta [26, 45], which is a purely functional language in which atomic operations are extremely expensive: compilation of files, archiving of libraries. The application is best illustrated by the following example:

$$\text{let } f\ x = \text{if } isC(x) \text{ then } Ccompile(x) \text{ else } M3compile(x)$$

$$\text{in } f(my\_file)$$

The function isC checks whether its argument file is a C file. The function $Ccompile$ is a function that calls the C compiler on its argument and the function $M3compile$ is a function that calls the Modula3 compiler on its argument.

If the free variable my_file is a C file then the above program need not be re-executed if the function $M3compile$ is bound to a different function, $M3compile_{new}$, that calls a

newer version of the Modula3 compiler. To keep re-execution of an altered program to a minimum, we need to isolate the set of subterms which 'need' to be evaluated in the course of the computation. If changes are made to subterms which were not 'needed' then re-execution is not required.

The language addressed by Abadi et al is the pure λ-calculus. The technique used to specify 'needed' subterms is that of a λ-calculus with *holes*. A term with *holes* is called a prefix. Prefixes and contexts are specified by the following grammar:

$$a \quad ::== \quad \_ \qquad\qquad\qquad C[\,] \quad ::== \quad [\,]$$
$$| \quad x \qquad\qquad\qquad\qquad | \quad \lambda x.\, C[\,]$$
$$| \quad \lambda x.\, a \qquad\qquad\qquad | \quad C[\,](a)$$
$$| \quad a_1(a_2) \qquad\qquad\qquad | \quad a(C[\,])$$

The β reduction rule is given by,

$$(\lambda x.\, b)a \;\rightarrow\; b\{a/x\}$$

There is also a congruence rule,

$$\text{if } a \;\rightarrow\; b \text{ then } C[a] \;\rightarrow\; C[b]$$

Prefixes have a partial order $\preceq$ defined on them. If a prefix a matches prefix b, except for the fact that corresponding to certain holes in a we have prefixes in b, then $a \;\preceq\; b$. Reduction on prefixes is performed by treating a hole $\_$ as a free variable.

Theorem 3.2.1 (Stability) *If a is a term, v is a term in a normal form, and $a \rightarrow^* v$, then there is a minimum prefix $a_0 \preceq a$ such that $a_0 \rightarrow^* v$.*

The Stability Theorem is a specification of the minimum slice of a term and it establishes the fact that a minimum slice of a term is a well-defined concept. The authors then provide a technique for the computation of the minimum prefix/slice, through the use of a labelled λ-calculus. The set of labelled λ-terms a_L is given by the following grammar:

$$a_L,\, b_L \quad ::== \quad \_ \mid x \mid \lambda x.\, a_L \mid a_L\,(b_L)$$
$$| \quad l : a_L \quad l \in L$$

where L is the set of labels.

Reduction in the labelled calculus requires another rule,

$$(l : b)(a) \rightarrow l : b(a)$$

Given a term a, let us label every subterm in a by a distinct label to obtain a labelled term a'. Let $a' \rightarrow^* v'$, where v' is a labelled term in normal form. Let L_0 be the labels syntactically contained in v'. Let $G(a)$ be the prefix obtained from a by replacing the subterms, whose labels are not included in L_0, by a hole. It is shown, by a Church-Rosser theorem [12], that G is a well-defined function on normalising terms. The following theorem shows that evaluation in this labelled calculus computes the minimum prefix of a term.

Theorem 3.2.2 *If a is a term, v is a term in normal form, $a \rightarrow^* v$, and $G(a) \preceq b$ then $b \rightarrow^* v$.*

The above theorem states that for a reduction $a \rightarrow^* v$ we can make a cache entry $(G(a), v)$. Before evaluating a term b, we need to check whether $A \preceq b$ for some cache entry (A, v). If so, we return the value v instead of performing the computation.

The analysis above was for arbitrary strong reductions. Instead, if we use the evaluation strategy, call-by-value, then we must take non-termination into account. A subterm which is not 'needed' can no longer be replaced by a hole occurrence. This is because a term which matches such a prefix can have a non-terminating computation, at the position corresponding to the hole occurrence. Hence, every subterm that is executed is needed. A restricted version of the β rule is used:

$$(\lambda x.\, b)\, v \rightarrow b\{v/x\}$$

where v includes terms of the x, $x\,(a_1)\ldots(a_n)$, or $\lambda x.\, a$.

Since every subterm that is executed is needed, we the following additional rule:

$$(\lambda x.\, b)\,(l : a) \rightarrow l : ((\lambda x.\, b)\,(a))$$

The results presented by Abadi et al are very similar to our own research for purely functional programs. Our setting is actually much simpler since we use a deterministic

call-by-value interpreter, in contrast to the general setting of arbitrary reductions. We do not need to prove a Church-Rosser Theorem or a Stability Theorem. Since the analysis is directed towards caching of computation, presence of side-effects, like exceptions, assignments or non-termination, causes a serious interference because of loss of referential transparency.

A labelled λ-calculus with a generalised definition of what it means for subterms to be needed in a computation is given by Gandhe et al [24]. But their definitions and characterisations are rooted in an undecidable concept: solvability.

Chapter 4

Syntax and Semantics of LML

The higher-order programming language we are going to use in this proposal is Standard ML(SML). The entire language has a formal definition presented by Milner et al in [48]. SML consists of a lower level called the *Core* language, a middle level concerned with programming-in-the large called *Modules* and a very small upper level called *Programs*. The execution of an SML declaration consists of three separate phases: *parsing*, *elaboration* and *evaluation*. Specification of *parsing* involves specification of syntax for the language. *Elaboration*, the static phase, determines whether the declaration is well-typed and well-formed. Specification of *evaluation* involves specifying the dynamic semantics for the language. With three levels in the structure of the language and three phases in the execution, the specification of the complete language can be broken into nine separate sections.

In this chapter and for most of the proposal we will be concerned with the *Core* language. Of the three phases in the execution of a term in the *Core* language: *parsing*, *elaboration* and *evaluation*, we will completely skip the *elaboration* phase. This is because most of the techniques developed in this proposal apply equally to both well-typed and untyped SML programs. For the specification of the dynamic semantics, in the evaluation phase, we will not be using SML syntax: we will be using a skeletal language, **LML**, whose grammar we are going to define and which essentially captures most syntactic constructs in Core-SML.

4.1 Language Syntax

The skeletal language under consideration, LML, is given by the grammar in Fig 4.1.

$$
\begin{array}{lll}
e & ::= & x \\
  & | & C(e_1, \ldots, e_n) \\
  & | & \lambda x.\, e \\
  & | & \text{letrec } f(x) = e_1 \text{ in } e_2 \\
  & | & e_1 e_2 \\
  & | & \text{Op}(e_1, \ldots, e_n) \\
  & | & \text{case}(e_1,\, C(x_1, \ldots, x_n) \Rightarrow e_2,\, y \Rightarrow e_3) \\
  & | & !\, e_1 \\
  & | & \text{ref}\, e_1 \\
  & | & e_1 := e_2 \\
  & | & \text{let exception D in } e_1 \\
  & | & e_1 \text{ handle } (D(x_1, \ldots, x_n) \Rightarrow e_2) \\
  & | & \text{raise } e_1
\end{array}
$$

Figure 4.1

The following are important points to be noted about the syntax:

- There are no constructor declarations in LML. Constructors have a static semantics in SML. Hence, issues involving local constructor declarations are relevant mostly to the elaboration phase of the language.

  LML expects every occurrence of a constructor to be saturated: every occurrence of a n-ary constructor must be an application to an n-tuple. This is unlike SML which allows a constructor to be passed around as a value/parameter. The restriction we place here is not serious: a simulation is possible.

- The grammar does not show the language as having boolean constants, natural numbers or real numbers. The set of natural numbers will be represented by infinitely many distinct nullary constructors. The set of booleans will be represented by two distinct nullary constructors.

  The atomic operators Op are assumed to operate on nullary constructors and return nullary constructors. This accommodates standard arithmetic and boolean operators

26

found in SML. It is to be noted that arithmetic operators in SML can raise exceptions.

Our approach to constructors and operators cannot accommodate the built-in datatypes like `string` and atomic string functions like `explode` and `implode`. We can give these operators a special status, like we give to ref or ! . For the theoretical sections we have decided to drop this datatype from the language.

- SML uses a binding construct `let` instead of the binding construct letrec used in LML. The `let` construct plays a very important role in static typing but for the evaluation phase, it is syntactic sugar except for its ability to provide binding for recursive function declarations. Hence, we use a letrec construct which can only bind function declarations.

  Our language does have a let construct. It is used exclusively for binding exception constructors. Exceptions in SML are *generative* in nature and hence their declaration is evaluated in the dynamic semantics. Reading the declaration of an exception as an SML declaration suggests that all our exceptions are nullary. But, this is not so: we leave out type declarations, as we are skipping the elaboration phase.

- The SML syntax provides us with the ability to explicitly declare mutually recursive functions. Our language provides no such facility. As is to be discussed later, this restriction results in a considerable simplification in the presentation of the dynamic semantics and proofs involving the dynamic semantics.

- SML allows for nested patterns in `case`, `handle` and function arguments. These are not allowed in LML to make things simple.

4.2   Dynamic Semantics

The specification of the dynamic semantics involves semantic objects called values. They are specified by the following grammar:

$$
\begin{aligned}
v \quad ::= \quad & \langle E,\, \lambda x\, e \rangle \\
| \quad & \langle E,\, f,\, \lambda x\, e \rangle \\
| \quad & C(v_1,\, \ldots,\, v_n) \\
| \quad & \beta \ \ \text{where } \beta \in \mathit{Loc} \\
| \quad & [\,\delta,\, (v_1,\, v_2,\, \ldots,\, v_n)\,] \ \ \text{where } \delta \in \Delta
\end{aligned}
$$

The environment E is a finite function which maps variables to values and, exception constructors to elements from a countably infinite set Δ.

$$
\begin{aligned}
E \quad ::= \quad & [\,] \\
| \quad & E[x \mapsto v] \\
| \quad & E[D \mapsto \delta]
\end{aligned}
$$

An exception packet is denoted by $\ll [\,\delta,\, (v_1,\, v_2,\, \ldots,\, v_n)\,] \gg$.

In the space of values, there are two kinds of closures: the standard function closure and the recursive function closure. The standard closure is denoted by $\langle E,\, \lambda x\, e \rangle$ and the recursive closure is denoted by $\langle E,\, f,\, \lambda x\, e \rangle$. The SML definition [48] does not make a distinction between the two kinds of closures. A closure, in the SML definition, has two environments instead of one: an environment for mutually recursive function definitions and an environment for other free variables. In the SML definition, whenever a closure is applied to an argument, the application rule unfolds the environment for mutually recursive functions once and adds it to the current environment, for mutually recursive functions. Our approach is to separate closures, for recursive function declarations, from other closures. Thus, in contrast to SML, we have two separate application rules in the dynamic semantics: one for the application of standard function closures to arguments and the other for the application of recursive function closures to arguments. In a recursive function closure, $\langle E,\, f,\, \lambda x\, e \rangle$, the second component f is the name of the recursive function, whose declaration generated this closure. There is a another approach possible. The definition of values and environments may be treated as *co-inductive* definitions, instead of *inductive* definitions. This would allow us to define the value of a recursively defined function to be its infinite unfolding. We could then use a single application, Rule 4.4. But then all our proofs would have to be co-inductions instead of inductions, as in [46].

The variable β is an element of the set *Loc*, the set of memory locations. In SML, exceptions are generative: every time an exception declaration is evaluated, the constructor D is mapped to a new unique element from the set Δ, the set of exception constructor values. An exception constructor D applied to a vector of arguments v_1, v_2, ... , v_n returns a value $[\delta, (v_1, v_2, \ldots, v_n)]$, if the current environment maps D to δ.

There is also a subtle difference with SML. All functions and constructors in SML, except obviously the pairing constructor, take in single arguments. We allow all our constructors, operators and exception constructors to take in multiple arguments.

Like the SML definition, we present the dynamic semantics of our language using natural semantics [54, 37]. The semantics presented below allows us to infer statements of the form:

$$S, Ex, E \vdash e \rightarrow v, S', Ex'$$

where S is the initial memory with which the evaluation of the term e begins. $Ex \subseteq \Delta$, denotes a set of elements already used in mappings of exception constructors. E is the initial environment. The value to which e evaluates to, is denoted by v. The final store, at the end of the computation, is given by S' and $Ex' \subseteq \Delta$ denotes the set of elements used in mappings of exception constructors, in the computation.

$$S_0, Ex_0, E[x \mapsto v] \vdash x \rightarrow v, S_0, Ex_0 \tag{4.1}$$

$$S_0, Ex_0, E \vdash \lambda x\, e \rightarrow \langle E, \lambda x\, e \rangle, S_0, Ex_0 \tag{4.2}$$

$$\frac{S_0, Ex_0, E[f \mapsto \langle E, f, \lambda x\, e_1 \rangle], \vdash e_2 \rightarrow v, S_1, Ex_1}{S_0, Ex, E \vdash \text{letrec } f(x) = e_1 \text{ in } e_2 \rightarrow v, S_1, Ex_1} \tag{4.3}$$

$$\frac{\begin{array}{c} S_0, Ex_0, E \vdash e_1 \rightarrow \langle E', \lambda x\, e \rangle, S_1, Ex_1 \\[4pt] S_1, Ex_1, E \vdash e_2 \rightarrow v_2, S_2, Ex_2 \\[4pt] S_2, Ex_2, E'[x \mapsto v_2] \vdash e \rightarrow v_3, S_3, Ex_3 \end{array}}{S_0, Ex_0, E \vdash e_1 e_2 \rightarrow v_3, S_3, Ex_3} \tag{4.4}$$

$$S_0 \, , \, Ex_0 \, , \, E \;\vdash\; e_1 \;\rightarrow\; \langle E' \, , \, f \, , \, \lambda x \, e \rangle \, , \, S_1 \, , \, Ex_1$$

$$S_1 \, , \, Ex_1 \, , \, E \;\vdash\; e_2 \;\rightarrow\; v_2 \, , \, S_2 \, , \, Ex_2$$

$$\frac{S_2 \, , \, Ex_2 \, , \, E'[f \mapsto \langle E' \, , \, f \, , \, \lambda x \, e \rangle \, , \, x \mapsto v_2] \;\vdash\; e \;\rightarrow\; v_3 \, , \, S_3 \, , \, Ex_3}{S_0 \, , \, Ex_0 \, , \, E \;\vdash\; e_1 e_2 \;\rightarrow\; v_3 \, , \, S_3 \, , \, Ex_3} \tag{4.5}$$

$$\frac{S_{i-1} \, , \, Ex_{i-1} \, , \, E \, , \;\vdash\; e_i \;\rightarrow\; v_i \, , \, S_i \, , \, Ex_i \quad i = 1 \ldots n}{S_0 \, , \, Ex_0 \, , \, E \;\vdash\; \mathrm{Op}(e_1 \, , \, \ldots \, , \, e_n) \;\rightarrow\; Op(v_1 \, , \, \ldots \, , \, v_n) \, , \, S_n \, , \, Ex_n} \tag{4.6}$$

The syntactic operator is denoted by Op. Its semantic counterpart is denoted by Op.

$$\frac{S_{i-1} \, , \, Ex_{i-1} \, , \, E \;\vdash\; e_i \;\rightarrow\; v_i \, , \, S_i \, , \, Ex_i \quad i = 1 \ldots n}{S_0 \, , \, Ex_0 \, , \, E \, , \, S_0 \, , \, L_0 \;\vdash\; \mathrm{C}(e_1 \, , \, \ldots \, , \, e_n) \;\rightarrow\; C(v_1 \, , \, \ldots \, , \, v_n) \, , \, S_n \, , \, Ex_n} \tag{4.7}$$

$$S_0 \, , \, Ex_0 \, , \, E \;\vdash\; e_1 \;\rightarrow\; C(v_1 \, , \, \ldots \, , \, v_n) \, , \, S_1 \, , \, Ex_1$$

$$\frac{S_1 \, , \, Ex_1 \, , \, E[x_1 \mapsto v_1 \, , \, \ldots \, , \, x_n \mapsto v_n] \;\vdash\; e_2 \;\rightarrow\; v \, , \, S_2 \, , \, Ex_2}{S_0 \, , \, Ex_0 \, , \, E \;\vdash\; \mathrm{case}(e_1 \, , \, \mathrm{C}(x_1 \, , \, \ldots \, , \, x_n) \Rightarrow e_2 \, , \, y \Rightarrow e_3) \;\rightarrow\; v \, , \, S_2 \, , \, Ex_2} \tag{4.8}$$

$$S_0 \, , \, Ex_0 \, , \, E \;\vdash\; e_1 \;\rightarrow\; C'(v_1 \, , \, \ldots \, , \, v_n) \, , \, S_1 \, , \, Ex_1 \quad C \neq C'$$

$$\frac{S_1 \, , \, Ex_1 \, , \, E[y \mapsto C'(v_1 \, , \, \ldots \, , \, v_n)] \;\vdash\; e_2 \;\rightarrow\; v \, , \, S_2 \, , \, Ex_2}{S_0 \, , \, Ex_0 \, , \, E \;\vdash\; \mathrm{case}(e_1 \, , \, \mathrm{C}(x_1 \, , \, \ldots \, , \, x_n) \Rightarrow e_2 \, , \, y \Rightarrow e_3) \;\rightarrow\; v \, , \, S_2 \, , \, Ex_2} \tag{4.9}$$

$$\frac{S_0 \, , \, Ex_0 \, , \, E \;\vdash\; e \;\rightarrow\; v \, , \, S_1 \, , \, Ex_1 \quad \beta \notin \mathrm{dom}(S_1)}{S_0 \, , \, Ex_0 \, , \, E \;\vdash\; \mathrm{ref}\, e \;\rightarrow\; \beta \, , \, S_1[\beta \mapsto v] \, , \, Ex_1} \tag{4.10}$$

$$S_0 \, , \, Ex_0 \, , \, E \;\vdash\; e_1 \;\rightarrow\; \beta \, , \, S_1 \, , \, Ex_1$$

$$\frac{S_1 \, , \, Ex_1 \, , \, E \;\vdash\; e_2 \;\rightarrow\; v \, , \, S_2 \, , \, Ex_2}{S_0 \, , \, Ex_0 \, , \, E \;\vdash\; e_1 := e_2 \;\rightarrow\; () \, , \, S_2[\beta \mapsto v] \, , \, S_2 \, , \, Ex_2} \tag{4.11}$$

$$\frac{S_0 \, , \, Ex_0 \, , \, E \;\vdash\; e_1 \;\rightarrow\; \beta \, , \, S_1 \, , \, Ex_1 \quad \text{where } S_1(\beta) = v}{S_0 \, , \, E_0 \, , \, E \;\vdash\; !e_1 \;\rightarrow\; v \, , \, S_1 \, , \, Ex_1} \tag{4.12}$$

The rules involving exceptions are given below.

$$\frac{S_{i-1} \, , \, Ex_{i-1} \, , \, E[D \mapsto \delta] \;\vdash\; e_i \;\rightarrow\; v_i \, , \, S_i \, , \, Ex_i \quad i = 1 \ldots n}{S_0 \, , \, Ex_0 \, , \, E[D \mapsto \delta] \;\vdash\; \mathrm{D}(e_1 \, , \, \ldots e_n) \;\rightarrow\; [\delta \, , \, \overline{v_i}] \, , \, S_n \, , \, Ex_n} \tag{4.13}$$

$$\frac{S_0\,,\,Ex_0\,,\,E\,\vdash\,e\,\rightarrow\,[\delta\,,\,\overline{v_i}]\,,\,S_1\,,\,Ex_1}{S_0\,,\,Ex_0\,,\,E\,\vdash\,\mathrm{raise}\,e\,\rightarrow\,\ll[\delta\,,\,\overline{v_i}]\gg\,,\,S_1\,,\,Ex_1} \qquad (4.14)$$

$$\frac{S_0\,,\,Ex_0\,,\,E\,\vdash\,e_1\,\rightarrow\,v\,,\,S_1\,,\,Ex_1}{S_0\,,\,Ex_0\,,\,E\,\vdash\,e_1\,\mathrm{handle}\,(\mathrm{D}(x_1\,,\,\ldots\,,\,x_n)\Rightarrow e_2)\,\rightarrow\,v\,,\,S_1\,,\,Ex_1} \qquad (4.15)$$

$$\frac{S_0\,,\,Ex_0\,,\,E\,\vdash\,e_1\,\rightarrow\,\ll[\delta\,,\,\overline{v_i}]\gg\,,\,S_1\,,\,Ex_1 \quad \text{where}\;E(\mathrm{D})\,\equiv\,\delta'\neq\delta}{S_0\,,\,Ex_0\,,\,E\,\vdash\,e_1\,\mathrm{handle}\,(\mathrm{D}(x_1\,,\,\ldots\,,\,x_n)\Rightarrow e_2)\,\rightarrow\,\ll[\delta\,,\,\overline{v_i}]\gg\,,\,S_1\,,\,Ex_1} \qquad (4.16)$$

$$\frac{\begin{array}{c}S_0\,,\,Ex_0\,,\,E\,\vdash\,e\,\rightarrow\,\ll[\delta\,,\,\overline{v_i}]\gg\,,\,S_1\,,\,Ex_1 \\[4pt] S_1\,,\,Ex_1\,,\,E[x_1\mapsto v_1\,,\,\ldots\,,\,x_n\mapsto v_n]\,\vdash\,e_2\,\rightarrow\,v\,,\,S_2\,,\,Ex_2\end{array}}{S_0\,,\,Ex_0\,,\,E\,\vdash\,e\,\mathrm{handle}\,(\mathrm{D}(x_1\,,\,\ldots\,,\,x_n)\Rightarrow e_2)\,\rightarrow\,v\,,\,S_2\,,\,Ex_2} \qquad (4.17)$$

$$\text{where}\quad E(\mathrm{D})=\delta$$

$$\frac{S_0\,,\,Ex_0\cup\{\delta\}\,,\,E[D\mapsto\delta]\,\vdash\,e\,\rightarrow\,v_1\,,\,S_1\,,\,Ex_1 \quad \text{where}\;\delta\notin Ex_0}{S_0\,,\,Ex_0\,,\,E\,\vdash\,\mathrm{let\;exception\;D\;in}\,e\,\rightarrow\,v_1\,,\,S_1\,,\,Ex_1} \qquad (4.18)$$

More clauses need to be added to the rules, presented above, to complete the specification. A succinct presentation of these additional rules may be given, along the lines of [48], by the introduction of an *exception convention*.

$$\frac{\begin{array}{c}\ldots\,,\,E_1\,\vdash\,e_1\,\rightarrow\,v_1\,,\,\ldots \\ \ldots \\ \ldots \\ \ldots\,,\,E_n\,\vdash\,e_n\,\rightarrow\,v_n\,,\,S_n\,,\,Ex_n\end{array}}{\ldots\,,\,E,\,\vdash\,e\,\rightarrow\,v_n\,,\,,\,S_n\,,\,Ex_n}$$

$$\frac{\begin{array}{c}\ldots\,,\,E_1\,\vdash\,e_1\,\rightarrow\,v_1\,,\,\ldots \\ \ldots \\ \ldots\,,\,E_k\,,\,\vdash\,e_k\,\rightarrow\,\ll\ldots\gg\,,\,S_k\,,\,Ex_k\end{array}}{\ldots\,,\,E\,\vdash\,e\,\rightarrow\,\ll\ldots\gg\,,\,S_k\,,\,Ex_k}$$

$$(a) \hspace{6cm} (b)$$

By this convention, let an evaluation rule be of the form (a), with n antecedents. Then for every $k, 1\leq k\leq n$, such that e_k evaluates to an exception packet $\ll\ldots\gg$ and $\forall j,\,1\leq j<k$, e_j evaluates to a value, we add another another rule of the form (b).

Chapter 5

Slicing Purely Functional Programs

The first-order programming language **L**, discussed in Chapter 2, was a statement-based language, i.e. a program written in **L** consisted of a sequence of statements. In contrast, a higher-order programming language like Standard ML(SML), [51, 48], is an expression-based language. For such languages, the task of generating executable dynamic slices is far from over, even after the set of subexpressions that 'contribute' to the value, returned by a program, have been isolated. This is because the deletion of an arbitrary set of subexpressions no longer leaves behind a legal expression that is executable. Thus the concepts associated with the slicing of first-order programs do not carry over, as is, into the domain of higher-order programs.

Interprocedural slicing of first-order programs was first investigated by Weiser [67] in his seminal paper. This analysis was greatly improved by Horwitz et al [31, 32]. The analysis developed by Horwitz et al is essentially an evaluation of an attribute grammar constructed from the procedure call-graph of a program. For a first-order program, the procedure call-graph can be trivially constructed from the parse tree of a program. For a higher-order program, the procedure call-graph cannot be statically constructed because we need to know about the bindings of formal parameters to actual parameters.

In this chapter, formal definitions of dynamic slices and associated algorithms for their computation are presented with respect to the operational semantics for the language.

Using an operational definition for dynamic slices, makes the proof of correctness of the algorithm, computing dynamic slices, much easier.

5.1 Formal Definition of Dynamic and Relevant Slices

The concepts associated with the definition of program slices for first-order programs, under a given operational semantics, have counterparts in the higher-order case:

- Corresponding to an execution trace, we have a proof tree of the evaluation of a program, under natural semantics.

- We prefer to use a *fixed slicing criteria*: the value returned by the program. This is similar to the criteria used in the formal definitions provided in Section 2.1. After we have built the required machinery, we will allow for a more general slicing criteria, similar to [67].

Typically, a statement in a first-order imperative program is referred to by an associated statement number. Similarly, a subterm in a higher-order program will be referred to by an associated label. Given a parse tree of a program, an initial assignment of labels to subexpressions/subtrees can be done with the use of *occurrences*, as described in [15].

Definition: For every natural number k, let s_k be a function that maps any tree, $op(t_1, \ldots, t_k, \ldots, t_n)$ to t_k. An **occurrence** is defined as any function obtained by composing an arbitrary number of such functions s_i.

All programs considered, hence, will be assumed to have their all subterms labelled. Terms will no longer be considered in isolation of their labels. Henceforth, all terms will be represented as a label and term separated by a ":" . Computation of the slice of a term is a computation collecting labels. This chapter deals with a purely functional language: a language without exceptions and assignments. The grammar given below defines legal *labelled* terms.

$$
\begin{array}{lll}
M & ::= & l : e \\[2mm]
e & ::= & x \\[2mm]
& | & C(M_1\,,\,\ldots\,,\,M_n) \\[2mm]
& | & \lambda x.\,M \\[2mm]
& | & \text{letrec } f(x) = M_1 \text{ in } M_2 \\[2mm]
& | & M_1 M_2 \\[2mm]
& | & Op(M_1\,,\,\ldots\,,\,M_n) \\[2mm]
& | & case(M_1\,,\,C(x_1\,,\,\ldots\,,\,x_n) \Rightarrow M_2\,,\,y \Rightarrow M_3)
\end{array}
$$

Notation: Terms having labelled roots, and all their subterms labelled, will be represented by variables M, N, ...

Terms *not* having labelled roots, but having all theirs subterms labelled, will be represented by variables e, f, ...

To ensure that the deletion of an arbitrary subterm leaves behind a legal expression, we introduce a new constant $-$ into the language and define deletion of a subterm by substitution with $-$. We provide $-$ with the same operational semantics as a *skip* instruction/*no-op* term. Rules involving the constant $-$ are given in Table 5.1. These rules are termed as the empty rules of the language.

$$
E \vdash l : - \;\rightarrow\; -
$$

$$
\frac{E \vdash M_1 \;\rightarrow\; - \qquad E \vdash M_2 \;\rightarrow\; v}{E \vdash l : M_1 M_2 \;\rightarrow\; -}
$$

$$
\frac{E \vdash M_{i_0} \;\rightarrow\; - \qquad E \vdash M_i \;\rightarrow\; v_i\,,\; i \in \{1 \ldots n\} - \{i_0\}}{E \vdash l : Op(M_1\,,\,\ldots\,,\,M_n) \;\rightarrow\; -}
$$

$$
\frac{E \vdash M_1 \;\rightarrow\; -}{E \vdash l : case(M_1\,,\,C(x_1\,,\,\ldots\,,\,x_n) \Rightarrow M_2\,,\,y \Rightarrow M_3) \;\rightarrow\; -}
$$

Table 5.1: Empty Rules

Unless $-$ occurs in a position, where the internal structure of the term needs to be examined, e.g. in the predicate component of an if-then-else, the rules remain the same. Thus ensuring that $-$ is a *no-op* term.

Notation: For any label l and terms M and e, $M[l/e]$ denotes the term obtained, from M, by replacing the subterm labelled l, in M, by the term e.

Definition: Let $\vdash (l_0 : e) \to v$. The set of labels $\mathcal{L} \subseteq \Lambda(l_0 : e)$ defines a **dynamic slice** of $(l_0 : e)$, if for $\mathcal{L}' \equiv \Lambda(l_0 : e) - \mathcal{L}$, $\vdash (l_0 : e)[\mathcal{L}'/-] \to v[\mathcal{L}'/-]$

A program slice, defined in Section 2.1, is a sub-program, of the original program, that is executable on the standard interpreter. As per the definition above, a sliced version of a higher-order program is no longer legal under the original syntax: it is the original program with a set of subterms replaced by $-$. While this breaks away from the traditional concept of an executable slice, it is not completely novel. The slice of term, as defined by Field and Tip [22], is not a term: it is a context. The sliced version of a program is no longer executable on the standard interpreter. To execute a sliced version of a program, the standard interpreter needs to be augmented with the set of rules associated with $-$. Henceforth, whenever we talk about an executable slice, we actually mean executable on the standard interpreter, augmented by the set of rules for $-$. Later on, we will briefly discuss the construction of slices that can be executed by the standard interpreter.

The definition of a relevant slice, in Chapter 1, was with respect to a restricted definition of program alteration. As shown in [7], even for simple programs, with such a restricted definition of program alteration, the computation of a relevant slice involves performing data-flow analysis for reaching definitions. Term alteration will be defined as a substitution of a subterm by an arbitrary term, or the deletion of a subterm, i.e. substitution by $-$. This necessarily means that the evaluation of an altered term may fail to terminate. Informally, if a subterm does not belong to a relevant slice then any alteration to this subterm does not change the value returned by the program. But this assertion is now true modulo termination. If the altered program terminates then it returns the same value as the original program.

A nullary constructor is defined to be a first-order value.

Definition: Let $\vdash (l_0 : e_0) \rightarrow v$, where v is a first-order value. The set of labels $\mathcal{L} \subseteq \Lambda(l_0 : e_0)$ defines a **relevant slice** of $(l_0 : e_0)$, if for any $\vec{l} \subseteq (\Lambda(l_0 : e_0) - \mathcal{L})$, and any substitution[1] $[\vec{l}/\vec{e}]$, if $\vdash (l_0 : e_0[\vec{l}/\vec{e}]) \rightarrow v'$ then $v' \equiv v$.

Unlike the definition of dynamic slices, the above definition of relevant slices is restricted to programs which return first-order values. This is because syntactic identity does not hold when closures are returned. But, a program returning a first-order value may have sub-terms computing higher-order values. Hence, we necessarily need to talk about higher-order values. In the following sections we are going to develop the machinery to talk about higher-order values.

5.2 A Natural Semantics for Computation of Slices

To ensure that a variable, in a first-order program, takes on a specific value we need to ensure that certain assignment statements are executed. To ensure that a specific statement is executed, we need to ensure that predicates on which it is control dependent [18] evaluate to the same value as in the original execution. For a block-structured first-order program, control dependency on a predicate can be trivially identified, while for an arbitrary first-order program post-dominator analysis [43] is required. As shown in [18], control dependency analysis for first-order programs can be statically performed. As discussed in Section 2.2, control-dependency information, for a first-order program, can be easily computed given the execution trace $\mathcal{T}$ of a program. Unlike first-order programs, control flow in a higher-order program depends on the binding of formal higher-order variables to actual functions. Thus, to compute control-dependency information from the execution trace, in this case a proof tree, we need to pass around the information as a parameter. Hence, the simplest way to specify an algorithm to compute dynamic slices is to provide a modified operational semantics.

Specifying Dynamic Slices As a Proof System

In [48], Milner et al present the semantics of SML as a natural deduction proof system. In Fig 5.2, we use a similar proof system to specify dynamic slices.

---

[1]The terms substituted for $\vec{l}$ may be $\bot$. $(l_0 : e[\vec{l}/\vec{e}])$ must be closed.

If $\Vdash M \to V, L$ then L is a dynamic slice of M. To prove this, we need a stronger induction hypothesis because of the presence of free variables and environments. Actually, the proof system is a specification of the minimum dynamic slice. It is easy to show this, once the main lemma has been established. The set of labels L, associated with values in the semantics, does not denote the entire set of dependencies required for the computation of the value. In fact, it is actually a subset of the entire set. This is good enough, in the case of purely functional programs. But fails in the presence of assignments and exceptions.

The set of values V computed by the operational semantics is specified by the grammar given below. L is a set of labels.

$$V \quad ::= \quad \langle F, \lambda x\, M \rangle$$
$$| \quad \langle F, f, \lambda x\, M \rangle$$
$$| \quad C((V_1, L_1), \ldots, (V_n, L_n))$$

The environment F is a map, $Var \to V * \mathcal{P}(L)$

$$F \quad ::= \quad [\,]$$
$$| \quad F[x \mapsto (V, L)]$$

Definition: A substitution function is defined on the values computed by the natural semantics in Table 5.2

$$(V, L)[\mathcal{L}/-] =$$
$$\quad \text{if } (L \cap \mathcal{L}) \neq \emptyset$$
$$\quad \text{then } -$$
$$\quad \text{else } \text{ case } V \text{ of}$$
$$\qquad \langle F, \lambda x\, M \rangle \ \Rightarrow \ \langle F[\mathcal{L}/-], \lambda x\, M[\mathcal{L}/-] \rangle$$
$$\qquad \langle F, f, \lambda x\, M \rangle \ \Rightarrow \ \langle F[\mathcal{L}/-], f, \lambda x\, M[\mathcal{L}/-] \rangle$$
$$\qquad C((V_1, L_1), \ldots, (V_n, L_n)) \ \Rightarrow \ C((V_1, L_1)[\mathcal{L}/-], \ldots, (V_n, L_n)[\mathcal{L}/-])$$
$$\text{and } (F[x \mapsto (V, L)])[\mathcal{L}/-] \ = \ (F[\mathcal{L}/-])[x \mapsto (V, L)[\mathcal{L}/-]]$$
$$\text{and } [\,][\mathcal{L}/-] \ = \ [\,]$$

The substitution function attempts to capture the intuition that for a tuple (V, L),

$$F[x \mapsto (V, L)] \;\Vdash\; l : \; x \;\to\; V \,,\; L \cup \{l\} \tag{5.1}$$

$$F \;\Vdash\; l : \; \lambda x\, M \;\to\; \langle F \,,\; \lambda x\, M \rangle \,,\; \{l\} \tag{5.2}$$

$$\frac{F[f \mapsto (\langle F \,,\; f \,,\; \lambda x\, M_1 \rangle \,,\; \{l\})] \;\Vdash\; M_2 \;\to\; V \,,\; L}{F \;\Vdash\; l : \; \text{letrec } f(x) = M_1 \text{ in } M_2 \;\to\; V \,,\; L \cup \{l\}} \tag{5.3}$$

$$\frac{F \;\Vdash\; M_1 \;\to\; \langle F_1 \,,\; \lambda x\, M \rangle \,,\; L_1 \quad F \;\Vdash\; M_2 \;\to\; V_2 \,,\; L_2 \qquad F_1[x \mapsto (V_2, L_2)] \;\Vdash\; M \;\to\; V_3 \,,\; L_3}{F \;\Vdash\; l : \; M_1 M_2 \;\to\; V_3 \,,\; L_1 \cup L_3 \cup \{l\}} \tag{5.4}$$

$$\frac{F \;\Vdash\; M_1 \;\to\; \langle F_1 \,,\; f \,,\; \lambda x\, M \rangle \,,\; L_1 \quad F \;\Vdash\; M_2 \;\to\; V_2 \,,\; L_2 \qquad F_1[f \mapsto (\langle F_1 \,,\; f \,,\; \lambda x\, M \rangle \,,\; L_1) \,,\; x \mapsto (V_2, L_2)] \;\Vdash\; M \;\to\; V_3 \,,\; L_3}{F \;\Vdash\; l : \; M_1 M_2 \;\to\; V_3 \,,\; L_1 \cup L_3 \cup \{l\}} \tag{5.5}$$

$$\frac{F \;\Vdash\; M_i \;\to\; V_i \,,\; L_i \,,\; i = 1 \ldots n}{F \;\Vdash\; l : \; \text{Op}(M_1 \,,\; \ldots \,,\; M_n) \;\to\; Op(V_1 \,,\; \ldots \,,\; V_n) \,,\; \{l\} \cup \bigcup_{i=1}^n L_i} \tag{5.6}$$

$$\frac{F \;\Vdash\; M_i \;\to\; V_i \,,\; L_i \,,\; i = 1 \ldots n}{F \;\Vdash\; l : \; \text{C}(M_1 \,,\; \ldots \,,\; M_n) \;\to\; C((V_1 \,,\; L_1) \,,\; \ldots \,,\; (V_n \,,\; L_n)) \,,\; \{l\}} \tag{5.7}$$

$$\frac{F \;\Vdash\; M_1 \;\to\; C((V_1 \,,\; L_1) \,,\; \ldots \,,\; (V_n \,,\; L_n)) \,,\; L \qquad F[x_1 \mapsto (V_1 \,,\; L_1) \,,\; \ldots \,,\; x_n \mapsto (V_n \,,\; L_n)] \;\Vdash\; M_2 \;\to\; V \,,\; L'}{F \;\Vdash\; l : \; \text{case}(M_1 \,,\; \text{C}(x_1 \,,\; \ldots \,,\; x_n) \Rightarrow M_2 \,,\; y \Rightarrow M_3) \;\to\; V \,,\; L \cup L' \cup \{l\}} \tag{5.8}$$

$$\frac{F \;\Vdash\; M_1 \;\to\; C'((V_1 \,,\; L_1) \,,\; \ldots \,,\; (V_n \,,\; L_n)) \,,\; L \quad C \neq C' \qquad F[y \mapsto C'((V_1 \,,\; L_1) \,,\; \ldots \,,\; (V_n \,,\; L_n))] \;\Vdash\; M_2 \;\to\; V \,,\; L'}{F \;\Vdash\; l : \; \text{case}(M_1 \,,\; \text{C}(x_1 \,,\; \ldots \,,\; x_n) \Rightarrow M_2 \,,\; y \Rightarrow M_3) \;\to\; V \,,\; L \cup L' \cup \{l\}} \tag{5.9}$$

Table 5.2 : Specifying Dynamic Slices for Functional Programs

the set of labels L contributed towards the computation that led to the value V. Hence, if any $l \in L$ is substituted by $-$ then the computation terminates returning $-$.

Rule 5.4 illustrates the way we capture the intuition behind labels 'contributing' towards a computation. For the evaluation of $l : M_1 M_2$ the set of labels contributing towards the evaluation of M_1 to a value must be included. The set of labels contributing towards the evaluation of the argument M_2 is not explicitly included, as the argument to the function may not be explicitly used. Instead the tuple (V_2 , L_2) is bound in the environment. It may be the case that $L_2 \nsubseteq L_3$ but the tuple (V_2 , L_2) is a part of the closure V_3. If this is the case, then the set of labels L_2 may subsequently form a part of the dynamic slice.

Lemma 5.2.1 *For any set of labels $\mathcal{L}$, if $F \Vdash (l : e) \rightarrow (V , L)$*
then $F[\mathcal{L}/-] \vdash (l : e)[\mathcal{L}/-] \rightarrow (V , L)[\mathcal{L}/-]$.

Proof: The proof is constructed by induction on the height of the proof tree. The cases discussed in the proof make the assumption that $l \notin \mathcal{L}$. If this were not the case then the axiom $F[\mathcal{L}/-] \Vdash l : - \rightarrow -$, provides the requisite proof.

Rule(5.4) If $l \notin \mathcal{L}$ and $\mathcal{L} \cap L_1 = \emptyset$ then, by induction,

$\quad F[\mathcal{L}/-] \vdash M_1[\mathcal{L}/-] \rightarrow \langle F_1[\mathcal{L}/-] , \lambda x\, M[\mathcal{L}/-] \rangle$.

$\quad$ By induction, $F[\mathcal{L}/-] \vdash M_2[\mathcal{L}/-] \rightarrow (V_2 , L_2)[\mathcal{L}/-]$.

$\quad F_1[\mathcal{L}/-][x \mapsto (V_2, L_2)[\mathcal{L}/-]] \vdash M[\mathcal{L}/-] \rightarrow (V_3 , L_3)[\mathcal{L}/-]$.

$\quad$ Hence, $F[\mathcal{L}/-] \vdash (l : M_1 M_2)[\mathcal{L}/-] \rightarrow (V_3 , L_3)[\mathcal{L}/-]$.

$\quad$ As $(L_1 \cup \{l\}) \cap \mathcal{L} = \emptyset$, $(L_1 \cup L_3 \cup \{l\}) \cap \mathcal{L} = \emptyset$ iff $L_3 \cap \mathcal{L} = \emptyset$.

$\quad$ Hence, $(V_3 , L_3)[\mathcal{L}/-] = (V_3 , L_1 \cup L_3 \cup \{l\})[\mathcal{L}/-]$.

$\quad$ If $l \notin \mathcal{L}$ but $\mathcal{L} \cap L_1 \neq \emptyset$ then, by induction, $\quad F[\mathcal{L}/-] \vdash M_1[\mathcal{L}/-] \rightarrow -$.

$\quad$ Hence, $F[\mathcal{L}/-] \vdash (l : M_1 M_2)[\mathcal{L}/-] \rightarrow -$.

Rule(5.6) If $l \notin \mathcal{L}$ and $\mathcal{L} \cap \cup_{i=1}^{n} L_i = \emptyset$ then, since operators can only be applied to nullary constructors, by induction, we have, $\quad F[\mathcal{L}/-] \vdash M_i[\mathcal{L}/-] \rightarrow V_i$.

$\quad$ Hence, $F[\mathcal{L}/-] \vdash l : \mathrm{Op}(M_1 , \ldots , M_n)[\mathcal{L}/-] \rightarrow Op(V_1 , \ldots , V_n)$.

$\quad$ If, for some i, $\mathcal{L} \cap L_i \neq \emptyset$ then, by induction, $F[\mathcal{L}/-] \vdash M_i[\mathcal{L}/-] \rightarrow -$.

Since operators are strict in all their arguments,
$$F[\mathcal{L}/-] \vdash l : \mathrm{Op}(M_1,\ldots,M_n)[\mathcal{L}/-] \rightarrow -.$$

Rule(5.8) If $l \notin \mathcal{L}$ and $\mathcal{L} \cap L = \emptyset$ then, by induction,
$$F[\mathcal{L}/-] \vdash M_1[\mathcal{L}/-] \rightarrow C((V_1,L_1)[\mathcal{L}/-],\ldots,(V_n,L_n)[\mathcal{L}/-]).$$
By induction, we have,

$$F[\mathcal{L}/-][x_1 \mapsto (V_1,L_1)[\mathcal{L}/-],\ldots,x_n \mapsto (V_n,L_n)[\mathcal{L}/-]] \vdash$$
$$M_2[\mathcal{L}/-] \rightarrow (V,L')[\mathcal{L}/-]$$

As $(L \cup \{l\}) \cap \mathcal{L} = \emptyset$, $(L \cup L' \cup \{l\}) \cap \mathcal{L} = \emptyset$ iff $L' \cap \mathcal{L} = \emptyset$.

Hence, $(V,L')[\mathcal{L}/-] = (V,L \cup L' \cup \{l\})[\mathcal{L}/-]$.

If $l \notin \mathcal{L}$ but $\mathcal{L} \cap L \neq \emptyset$ then, by induction, $F[\mathcal{L}/-] \vdash M_1[\mathcal{L}/-] \rightarrow -.$

Hence, $F[\mathcal{L}/-] \vdash (l : \mathrm{case}(M_1,C(x_1,\ldots,x_n) \Rightarrow M_2,y \Rightarrow M_3))[\mathcal{L}/-] \rightarrow -.$

$\square$

Theorem 5.2.1 *If* $\emptyset \Vdash M \rightarrow (V,L)$ *then* L *is the minimum dynamic slice, i.e. for any dynamic slice* L', *of the evaluation of* M, $L \subseteq L'$.

Proof: If $\mathcal{L} = \Lambda(M) - L$ then $\mathcal{L} \cap L = \emptyset$. Thus, by Lemma 5.2.1, we have
$\vdash M[\mathcal{L}/-] \rightarrow V[\mathcal{L}/-]$. Hence, by definition, L is a dynamic slice.

Let L' be any dynamic slice. Let $\mathcal{L}' = \Lambda(M) - L'$. By Lemma 5.2.1, for any $l_0 \in L$, if
$l_0 \in \mathcal{L}'$ then $\vdash (l : M)[\mathcal{L}'/-] \rightarrow -$. Hence, for any dynamic slice L', $L \subseteq L'$. Thus L
defines the minimum dynamic slice.

$\square$

Relevant Slices and Dynamic Slices co-incide

To prove the co-incidence of dynamic and relevant slices, we need to prove that, if
$\Vdash M \rightarrow V,L$,   where V is a first-order value, then for any $\vec{l} \subseteq (\Lambda(M) - L)$ and any
$\vec{e}$, if the evaluation of $M[\vec{l}/\vec{e}]$ terminates then $\vdash M[\vec{l}/\vec{e}] \rightarrow V$. Having shown this, it is
trivial to show that this is the minimum relevant slice. This is because for any $q \in L$, we
already know that $M[q/-]$ evaluates to $-$.

Though relevant slices are defined for first-order programs, we need to talk about higher-order values, as the intermediate values computed may be higher-order. In Subsection 5.2, the value computed by a term, after the deletion of a set of subterms, was related to the original value by defining a substitution function $[\mathcal{L}/-]$. In the context of relevant slices, it is easier to relate the value, computed by the altered term, to the original value by a formal relation $\mathcal{R}$, instead of a function. This is because the value computed by the altered term is not known statically. The relation $\mathcal{R}$ is defined as follows:

- $(V\,,\,L)\ \mathcal{R}_{\vec{l}:\vec{e}}\ v\ \ \text{if}\ \ \vec{l}\cap L\neq\emptyset$

Else, if $\vec{l}\cap L=\emptyset$ then,

- $(\langle F_1\,,\,\lambda x\, M_1\rangle\,,\,L)\ \mathcal{R}_{\vec{l}:\vec{e}}\ \langle E_1\,,\,\lambda x\, M_2\rangle\ \ \ \text{iff}\ \ \ F_1\ \mathcal{R}_{\vec{l}:\vec{e}}\ E_1\ \ \text{and}\ \ M_1[\vec{l}/\vec{e}]\equiv M_2$

- $(\langle F_1\,,\,f\,,\,\lambda x\, M_1\rangle\,,\,L)\ \mathcal{R}_{\vec{l}:\vec{e}}\ \langle E_1\,,\,f\,,\,\lambda x\, M_2\rangle\ \ \ \text{iff}\ \ \ F_1\ \mathcal{R}_{\vec{l}:\vec{e}}\ E_1\ \ \text{and}\ \ M_1[\vec{l}/\vec{e}]\equiv M_2$

- $(C((V_1\,,\,L_1)\,,\,\ldots\,,\,(V_n\,,\,L_n))\,,\,L)\ \mathcal{R}_{\vec{l}:\vec{e}}\ C(v_1\,,\,\ldots\,,\,v_n)\ \ \ \text{iff}\ \ \ (V_i\,,\,L_i)\ \mathcal{R}_{\vec{l}:\vec{e}}\ v_i\ \ i=1\ldots n$

- $F[x\mapsto (V\,,\,L)]\ \mathcal{R}_{\vec{l}:\vec{e}}\ E[x\mapsto v]\ \ \text{iff}\ \ F\ \mathcal{R}_{\vec{l}:\vec{e}}\ E\ \text{and}\ (V\,,\,L)\ \mathcal{R}_{\vec{l}:\vec{e}}\ v$

- $[\,]\ \mathcal{R}_{\vec{l}:\vec{e}}\ [\,]$.

Lemma 5.2.2 *For any vector of labels $\vec{l}$, if $F\Vdash M\rightarrow (V\,,\,L)$ then if $(F\ \mathcal{R}_{\vec{l}:\vec{e}}\ E)$ and $E\vdash M[\vec{l}/\vec{e}]\rightarrow v$ then $(V\,,\,L)\,\mathcal{R}_{\vec{l}:\vec{e}}\,v$.*

Proof: The proof is by induction on the height of the proof tree. The proof is very similar in structure to the proof of Lemma 5.2.1. An outline of the proof, when Rule(5.4) is the last rule used, is given.

 Rule(5.4) If $l\in\vec{e}$ then the relation $\mathcal{R}_{\vec{l}:\vec{e}}$ trivially holds.

  If $l\notin\vec{l}$ but $\vec{l}\cap L_1\neq\emptyset$ then $\vec{l}\cap (L_1\cup L_3\cup\{l\})\neq\emptyset$ and hence, the relation $\mathcal{R}_{\vec{l}:\vec{e}}$ trivially holds.

  If $l\notin\vec{l}$ and $\vec{l}\cap L_1=\emptyset$ then, if the computation of $M_1[\vec{l}/\vec{e}]$ terminates, then, by induction,

  $E\vdash M_1[\vec{l}/\vec{e}]\rightarrow\langle E_1\,,\,\lambda x\, M[\vec{l}/\vec{e}]\rangle$ where $F_1\ \mathcal{R}_{\vec{l}:\vec{e}}\ E_1$.

  If the computation of $M_2[\vec{l}/\vec{e}]$ terminates, then, by induction, $(V_2\,,\,L_2)\ \mathcal{R}_{\vec{l}:\vec{e}}\ v_2$.

Hence, $F_1[x \mapsto (V_2 , L_2)] \, \mathcal{R}_{\vec{l}:\vec{e}} \, E_1[x \mapsto v_2]$.

If the computation of $M[\vec{l}/\vec{e}]$ terminates, i.e $E_1[x \mapsto v_2] \vdash M[\vec{l}/\vec{e}] \to v_3$

then, by induction, $(V_3 , L_3) \, \mathcal{R}_{\vec{l}:\vec{e}} \, v_3$. Thus, by definition, $(V_3 , L_1 \cup L_3 \cup \{l\}) \, \mathcal{R}_{\vec{l}:\vec{e}} \, v_3$.

$\square$

Theorem 5.2.2 *If $\Vdash M \to (V , L)$ then L is the minimum relevant slice of the first-order program M.*

The theorem is a trivial corollary to the above lemma.

5.3 Minimum Dynamic Slices and Call-By-Name Evaluation

As mentioned before, the intuition behind the formal definition of dynamic slices was the isolation of subterms which 'contribute' to the value computed by a term. For purely functional programs, there is a demand-driven evaluation strategy called the call-by-need lambda-calculus. This evaluation strategy can be implemented by a transition semantics, as in [10], or as a natural deduction proof system, as in [42]. Since such a strategy evaluates only what needs to be; the execution trace of such an evaluator should co-incide with the minimum dynamic slice. Indeed, this turns out to be so.

The natural semantics for lazy evaluation[42], involves a heap that is mutable. The mutable heap in the semantics, given in [42], is required to model sharing of evaluation: certain subterms are not re-evaluated in the course of the computation. But we are not concerned as to whether the same subterm gets re-evaluated multiple times. We are interested as to whether a subterm needs to get evaluated at least once. The call-by-name lambda calculus[25], is essentially the same as the call-by-need lambda calculus but without any sharing of evaluation. Hence, we use a natural semantics for the call-by-name lambda calculus and show that the execution trace associated with a term evaluating under this semantics exactly equals the minimum dynamic slice of the term. The proof rules collecting the execution trace under a call-by-name semantics are given in Table 5.3. If $\vdash_n M \Downarrow_L w$ then L represents the execution trace of the evaluation of M.

The set of call-by-name values w computed by the operational semantics is specified by the grammar given below.

$$w \quad ::= \quad \langle G \,,\, \lambda x \, M \rangle$$

$$| \quad \langle G \,,\, f \,,\, \lambda x \, M \rangle$$

$$| \quad C(\langle G_1 \,,\, M_1 \rangle \,,\, \dots \,,\, \langle G_n \,,\, M_n \rangle)$$

$$G \quad ::= \quad [\,]$$

$$| \quad G[x \mapsto \langle G \,,\, M \rangle]$$

To prove that the minimum dynamic slice co-incides with the execution trace of a call-by-name evaluator, we need to define a relation $\mathcal{E}$ between environments in the two semantics.

Definition:

- $F_0[x \mapsto (V \,,\, L)] \;\mathcal{E}\; G_0[x \mapsto \langle G \,,\, N \rangle]$

  iff $\;\; F_0 \;\mathcal{E}\; G_0 \;$ and $\; G \vdash_n N \; \Downarrow_L \; w \;$ where $\; V \;\mathcal{E}_{val} \; w.$

- $\langle F_1 \,,\, \lambda x \, M_1 \rangle \; \mathcal{E}_{val} \; \langle G_1 \,,\, \lambda x \, M_1 \rangle \;$ iff $\; F_1 \; \mathcal{E} \; G_1.$

- $\langle F_1 \,,\, f \,,\, \lambda x \, M_1 \rangle \; \mathcal{E}_{val} \; \langle G_1 \,,\, f \,,\, \lambda x \, M_1 \rangle \;$ iff $\; F_1 \; \mathcal{E} \; G_1.$

- $C((V_1 \,,\, L_1) \,,\, \dots \,,\, (V_n \,,\, L_n)) \; \mathcal{E}_{val} \; C(w_1 \,,\, \dots w_n) \;$ iff $\; [(V_i \,,\, L_i)] \; \mathcal{E} \; [\langle G_i \,,\, M_i \rangle].$

Lemma 5.3.1 *If* $\; F \Vdash M \rightarrow V \,,\, L \;$ *and* $\; F \; \mathcal{E} \; G \;$ *then* $\; G \vdash_n M \; \Downarrow_L \; w \;$ *where* $\; V \; \mathcal{E}_{val} \; w.$

Proof: The proof is by induction on the height of the proof tree. The important case are given below:

Rule(5.1) If $\; [x \mapsto (V \,,\, L)] \; \mathcal{E} \; [x \mapsto \langle E' \,,\, M \rangle] \;$ then, by definition, $E' \vdash_n M \; \Downarrow_L \; w,$
where $\; V \; \mathcal{E}_{val} \; w.$

Rule(5.4) By induction, $E \vdash_n M_1 \; \Downarrow_{L_1} \; \langle E_1 \,,\, \lambda x \, M \rangle \;$ where $F_1 \; \mathcal{E} \; E_1.$ Since $\; F \; \mathcal{E} \; E,$
by induction, if $F \Vdash M_2 \rightarrow (V \,,\, L) \;$ then $\; [x \mapsto (V \,,\, L)] \; \mathcal{E} \; [x \mapsto \langle E \,,\, M_2 \rangle].$
Hence, $\; F_1[x \mapsto (V \,,\, L)] \; \mathcal{E} \; E_1[x \mapsto \langle E \,,\, M_2 \rangle].$
Applying the induction hypothesis, we have, $E_1[x \mapsto \langle F \,,\, M_2 \rangle] \vdash_n M \; \Downarrow_{L_3} \; w$
where $V_3 \; \mathcal{E}_{val} \; w.$

$$\frac{G' \vdash_n M \Downarrow_L w}{G[x \mapsto \langle G' , M \rangle] \vdash_n l : x \Downarrow_{L \cup \{l\}} w} \tag{5.10}$$

$$G \vdash_n l : \lambda x\, M \Downarrow_{\{l\}} \langle G , \lambda x\, M \rangle \tag{5.11}$$

$$\frac{G[f \mapsto \langle G , f , \lambda x\, M_1 \rangle \vdash_n M_2 \Downarrow_L w}{G \vdash_n l : \text{letrec } f(x) = M_1 \text{ in } M_2 \Downarrow_{L \cup \{l\}} w} \tag{5.12}$$

$$\begin{array}{c} G \vdash_n M_1 \Downarrow_{L_1} \langle G' , \lambda x\, M \rangle \\[4pt] \dfrac{G'[x \mapsto \langle G , M_2 \rangle] \vdash_n M \Downarrow_{L_2} w}{G \vdash_n l : M_1 M_2 \Downarrow_{L_1 \cup L_2 \cup \{l\}} w} \end{array} \tag{5.13}$$

$$\begin{array}{c} G \vdash_n M_1 \Downarrow_{L_1} \langle G_1 , f , \lambda x\, M \rangle \\[4pt] \dfrac{G_1[f \mapsto \langle G_1 , f , \lambda x\, M \rangle , x \mapsto \langle G , M_2 \rangle] \vdash_n M \Downarrow_{L_3} w}{G \vdash_n l : M_1 M_2 \Downarrow_{L_1 \cup L_3 \cup \{l\}} w} \end{array} \tag{5.14}$$

$$\frac{G \vdash_n M_i \Downarrow_{L_i} w_i , \ i = 1 \ldots n}{G \vdash_n l : \text{Op}(M_1 , \ldots , M_n) \Downarrow_{\{l\} \cup \bigcup_{i=1}^n L_i} Op(w_1 , \ldots , w_n)} \tag{5.15}$$

$$G \vdash_n l : \text{C}(M_1 , \ldots , M_n) \Downarrow_{\{l\}} C(\langle G_1 , M_1 \rangle , \ldots , \langle G_n , M_n \rangle) \tag{5.16}$$

$$\begin{array}{c} G \vdash_n M_1 \Downarrow_L C(\langle G_1 , M_1 \rangle , \ldots , \langle G_n , M_n \rangle) \\[4pt] \dfrac{G[x_1 \mapsto (\langle G_1 , M_1 \rangle , \ldots , x_n \mapsto \langle G_n , M_n \rangle)] \vdash_n M_2 \Downarrow_{L'} w}{G \vdash_n l : \text{case}(M_1 , \text{C}(x_1 , \ldots , x_n) \Rightarrow M_2 , y \Rightarrow M_3) \Downarrow_{L \cup L' \cup \{l\}} w} \end{array} \tag{5.17}$$

$$\begin{array}{c} G \vdash_n M_1 \Downarrow_L C'(\langle G_1 , M_1 \rangle , \ldots , \langle G_n , M_n \rangle) \quad C \neq C' \\[4pt] \dfrac{G[y \mapsto C'(\langle G_1 , M_1 \rangle , \ldots , \langle G_n , M_n \rangle)] \vdash_n M_2 \Downarrow_{L'} w}{G \vdash_n l : \text{case}(M_1 , \text{C}(x_1 , \ldots , x_n) \Rightarrow M_2 , y \Rightarrow M_3) \Downarrow_{L \cup L' \cup \{l\}} w} \end{array} \tag{5.18}$$

Table 5.3 : Execution Under Call-By-Name Evaluation

Rule(5.8) By induction, $G \vdash_n M_1 \Downarrow_L C(\langle G_1, M_1 \rangle, \ldots, \langle G_n, M_n \rangle)$ where

$\quad [x_i \mapsto (V_i, L_i)] \; \mathcal{E} \; [x_i \mapsto \langle G_i, M_i \rangle]$, $i = 1 \ldots n$. Hence, by induction,

$\quad G[x_1 \mapsto (\langle G_1, M_1 \rangle, \ldots, x_n \mapsto \langle G_n, M_n \rangle] \vdash_n M_2 \Downarrow_{L'} w$ where $V \; \mathcal{E}_{val} \; w$.

$\square$

Theorem 5.3.1 *For any nullary constructor C_0, if $\Vdash M \to C_0, L$ then $\vdash_n M \Downarrow_L C_0$.*

Chapter 6

Static Analysis for Dynamic Slices

A denotational definition of static slices, for first-order programs, was presented in Section 2.1. A program in the language **L**, used in that section, consisted of a sequence of statements with free first-order variables. The computation of a static slice of a program p in **L**, wrt a slicing criterion, was essentially the process of isolation of a subprogram, whose behavior wrt the slicing criterion was identical to the original program, regardless of the instantiation of the free first-order variables. Since a control-flow graph [8] for a program in **L**, with free first-order variables, is statically constructible, a static slice (possibly) significantly smaller than the entire program can be computed.

If a program written, in our higher-order language, has free higher-order variables then the control flow becomes indeterminate. Consider the following program,

$$f\left(\lambda x \ \ldots\right)$$

If f is a free variable, instantiable to any arbitrary value, then we have no choice but to include its entire argument in the static slice. It is hopeless to expect anything but a gross over-approximation from any terminating algorithm trying to perform data-flow analysis [60, 61] on a higher-order program with free higher-order variables.

Hence, we decided to investigate whether there were terminating algorithms to compute approximations to the minimum dynamic slice of a closed higher-order functional program. More specifically,

Question: Is there a terminating algorithm, which given a closed term M as input,

computes a non-trivial dynamic slice of the term?

A trivial dynamic slice of a term M is its entire set of labels, $\Lambda(M)$.

6.1 Compiler Optimisations based on Analysis of Slices

Elimination of dead code is a standard optimisation performed by modern compilers for imperative languages. But compilers for higher-order programming languages perform only extremely naive kinds of dead code elimination. The principal benefit associated with the elimination of dead code in first-order imperative programs is the reduction of code size. As is to be illustrated shortly, for languages, with automatic garbage collection, lack of a good strategy for elimination of dead code may have more serious consequences than a larger code size: it may lead to greater heap space consumption.

The term *dead code* has been loosely used in compiler literature, [8], to refer to two distinct concepts:

1. Code that is never going to be *executed*: unreachable basic blocks.

   Consider, for example, the following statement,

$$\texttt{if (debug) then ...}$$

   If a copy propagation algorithm, run on the program, can determine that the flag `debug` has been set to `false` then the statements in the `then` arm of the program are never going to be executed. Hence, the entire `if` statement is considered dead code.

2. Code that is going to be executed but is not going to make any *contribution* to the final output of the program.

   Consider, for example, the following basic-block of a control-flow graph,

$$
\boxed{
\begin{array}{c}
\ldots \\
\texttt{v := } e \\
\ldots
\end{array}
}
$$

   If the variable `v` is not subsequently used in the basic block and a backward flow analysis can determine that `v` is not *live* at the end of this basic block then the above

assignment can be removed, as dead/useless code, without affecting the output of the program. This, of course, assumes that the evaluation of e has no side-effects.

We are now going to examine these concepts with respect to higher-order purely functional programs and two distinct operational semantics, *call-by-value* [53] and *call-by-need* [10, 42].

It is shown in Theorem 5.3.1 that the minimum dynamic slice exactly co-incides with the execution trace of a program evaluating under call-by-need semantics. Hence, dead code in lazy programs are subterms which are never evaluated, i.e., of the two distinct kinds of dead code elaborated earlier we only have the first kind.

Consider the program presented in Figure 6.1. Under a lazy semantics, the application

```
let  F = λg
        let    App = λF₁λf₁ F₁ f₁
               F₁ = λf₁λz f₁(f₁ 2)
               f₁ = λy y + 2
               w = g 1
               x = g 2
        in    if (w = 2) then  (App F₁ f₁ x)  else  3  end
    succ = λu u + 1
in  F succ  end
```

Figure 6.1

of F to *succ* results in creation of thunks for the let-bound variables App, F_1, f_1, w, x. The 'thunk' created for the variable x contains a pointer to the function *succ*. This thunk is never going to be evaluated but remains live till almost the end of the computation. Right after the evaluation of the expression $(w = 2)$ the function *succ* is garbage but cannot be collected as such because x is 'live' and contains a pointer to it.

According to Röjemo and Runciman [57], the biography of a typical cell in the heap includes four events: creation, first use, last use and destruction. A heap cell is said to be destroyed when it can be garbage collected. The phases between these events are called lag, use and drag respectively.

A heap cell, such as the thunk for the variable x in Figure 6.1, that is created but never subsequently used is referred to as being in the *void phase*. Such cells are retained in the heap , though not actually playing a role in the computation, because they form a part of the live graph. Refined garbage collection techniques, as discussed in [49, 23], which involve more than recursively following live pointers, can ascertain some of these thunks to be garbage. Reliance on such techniques makes a program less robust: a program with no space leaks may show one under a different runtime system.

A static analysis technique which can isolate subterms which are never going to be evaluated can improve the space-efficiency of a program executing under any garbage collector. This is because we can safely replace such subterms by fixed-size constants and still have the program return a value, identical to the value returned by the unoptimised program but consuming, possibly, much less space. Such a replacement strategy should prevent some cells in the void phase from being created at all and should reduce the drag phase of some heap cells. In the example program above, if we do not create the thunk for x we can remove the drag phase of *succ*.

If the above program is evaluated under a call-by-value semantics the variable x is going to be evaluated but is going to make no contribution to the value returned by the program. Hence, the subterm denoted by the variable x is dead code by the second criterion discussed above. Under call-by-value, a static analysis technique which replaces such sub-terms with constants may fail to generate a semantically equivalent program. This is because the optimised version of the program may terminate where the original program did not. If safety is equated with semantic equivalence then such a transformation is unsafe. From a pragmatic point of view, a more liberal definition of safety is good enough: if the original program terminates then the transformed program also terminates returning the same answer.

In [2], Abadi et al develop the concept of dynamic dependency analysis for λ-terms. The analysis developed here may be considered the static counterpart of such a dependency analysis.

An analysis technique which can statically compute a superset of the minimum dynamic slice of a purely functional program can thus be applied to compute a subset of the dead-code in a program.

49

6.2 Relation to Existing Work

Eliminating dead code is a standard optimisation in compilers for first-order imperative programming languages. The traditional approach, as described in [8], is to first perform *copy propagation* and then eliminate blocks, in the control-flow graph of the program, which are unreachable. A block in the control-flow graph is considered unreachable if any predicate, on which it is control dependent, can be statically analysed to evaluate to the negation of the value required to reach the block. The approach mentioned above attempts to discard blocks in the program which are never going to be executed. Copy-propagation is performed by forward analysis on the control-flow graph and has a well-defined counterpart for higher-order programs.

A more aggressive approach is to perform a *live-variable* analysis and then eliminate definitions for variables that are not live immediately outside the basic block in which they are defined. This approach discards code that would have been executed but would have made no contribution since the variable is dead after its definition. Live variable analysis is performed by backward analysis on the control-flow graph. Backward analysis does not seem to have a clear counterpart for higher-order programs.

Hughes [35] developed a technique for backward analysis of *first-order* functional programs, being evaluated in a lazy semantics. Given a closed first-order function f, of type $\tau_1 * \ldots * \tau_n \to \tau_0$ and abstract domains $\mathcal{A}_0, \ldots, \mathcal{A}_n$ for types $\tau_0, \ldots, \tau_n$, a backward analysis technique returns n functions, $f_i : \mathcal{A}_0 \to \mathcal{A}_i$. By choosing appropriate abstract domains and interpretations of primitives, a decidable analysis for isolating subterms, that are never going to be evaluated, can be performed. Hughes uses the term *absence analysis* for the technique. This technique is syntax-directed and does not seem to extend to higher-order functional programs because at every function call site, the analysis needs to know the exact function getting called.

Computable backward analysis has been extended to include higher-order functional programs in [17]. But [17] reverses abstract interpretation based on Scott-closed/Scott-open powerdomains [25]. The technique of reversing abstract interpretation has not been successfully applied to perform absence analysis.

There is an enormous body of research on abstract interpretation and its application

to strictness analysis, dating back to [50]. The results from that area are not applicable to the problem we seek to solve. This is because any computable strictness analysis technique computes a strict subset of the set of subterms which make a contribution to the computation. A solution to our problem involves a computation of a superset of such subterms. Besides, as shown in [59], any Mycroft-style strictness analysis completely ignores bindings of variables to constants.

6.3 A Set-Based Semantics

One of the simplest binding analysis techniques for call-by-value languages is a set-based analysis (SBA) technique developed by N. Heintze [30, 29, 28]. The development of a set-based analysis for a given natural semantics proceeds through the following stages:

A. Develop a set-based version of the operational semantics. The standard operational semantics for the language uses an environment, $E : Var \rightarrow Val$. The set-based semantics uses an environment, $\mathcal{E} : Var \rightarrow \mathcal{P}(Val)$ and evaluates a term into set of values, i.e. a subset of $\mathcal{P}(Val)$.

B. Define a property *safety* for set-based environments $\mathcal{E}$. Let $\vdash M \rightarrow v$, be a terminating computation, in the standard semantics, for a closed term M. Let $\mathcal{E}$ be any set-based environment *safe* wrt M. The safety property guarantees that there exists a computation $\mathcal{E} \vdash M \rightsquigarrow V$, in the set-based semantics such that $v \in V$.

C. For any given term M with a terminating computation, there exists a minimum set-based environment $\mathcal{E}_{min}$ that is safe.

D. The set-based approximation(sba) of a term is given by,

$$sba(M) \stackrel{\text{def}}{=} \{v \in V \mid \mathcal{E}_{min} \vdash M \rightsquigarrow V\}$$

E. Define a language of set-constraints and a syntax-directed translation from a closed term M to a set of constraints, such that a model of this set of constraints is a function that maps every subterm of M into a set of values that it might evaluate to.

51

It is shown that there is a minimum such model and it maps the term M exactly to the set $sba(M)$.

There is a polynomial time algorithm for the computation of the least model.

A natural semantics for the computation of dynamic slices, for purely functional programs, was presented in Table 5.2. Given any program $(l : e)$, if $\Vdash (l : e) \rightarrow (V, L)$ then $L \subseteq \mathcal{P}(Labels)$ is the minimum dynamic slice for the computation. We do not attempt to develop a set-based version of this semantics based on the steps elaborated above. This is because such a set-based semantics would be returning values which are elements of $\mathcal{P}(Val * \mathcal{P}(Labels))$. Our static analysis technique is built on a set-based semantics which incorporates the concept of demand into the semantics developed by Heintze.

Heintze's set-based analysis is decidable because it completely ignores inter-variable dependencies, and the fact that distinct evaluations of the same function, in distinct environments, return different values. If a specific occurrence of a subterm evaluates to a certain value v then, in a model for the set constraints, the subterm is mapped to a set of values containing v. But our natural semantics for the computation of slices throws away certain subcomputations because they make no contribution to the value that is returned. Hence, set-based analysis by Heintze, is definitely too much of an over-approximation for our purposes. Bindings coming out of subcomputations which make no contribution to the final value need to be thrown away. For example, consider the following program:

$$\text{let } f = \lambda x. \text{ if } (x = 1) \text{ then } 3 \text{ else } 4$$
$$\text{in } (\lambda y. f\,1)\,(f\,2) \text{ end} \tag{I}$$

The solution to the set-constraint problem, as described in [30], returns the fact that the variable x can be bound to the set $\{1, 2\}$ and the set of values, which can be returned by the program, equals $\{3, 4\}$.

What we are looking for, is an analysis technique which attempts to model the fact that a subterm, whose evaluation makes no contribution to the value returned, need not be evaluated. In the program (I), the subterm $(f\,2)$ does not contribute to the answer. Hence, we would like to have a set-based analysis which returns the variable x as being

bound to the set $\{1\}$, and the set of values which can be returned by the program as $\{3\}$.

The fundamental premise of our analysis is to completely ignore inter-variable dependencies. Hence, even on incorporating the concept of demand, we will conclude that certain sub-terms need to be evaluated even though they actually do not. For example, consider the program in Figure 6.2. In the call, `F (G f1) 2 3`, the value `z = f x` needs to be evaluated. Since evaluation contexts are completely ignored, the technique assumes that the value `z = f x` needs to be evaluated in the call, `F (H f2) 0 1`. Hence, the technique must infer that the subterm `(H f2)` needs to be evaluated, even though it is not needed in the computation.

```
let fun F  f x y = let val z = f x
                    in
                    if (y=1) then 90 else z
                    end
in
 F (G f1) 2 3  + F (H f2) 0 1
end
```

Figure 6.2

6.3.1 A Set-Based Semantics Incorporating Demand

As mentioned in the section above, the set-based semantics developed in [30] uses a global set-based environment $\mathcal{E} : Var \to \mathcal{P}(Val)$, which maps bound variables to a set of values. In addition to a global set-based environment $\mathcal{E}$, we introduce a global boolean environment $\mathcal{F} : Var \to (Bool \equiv \{t, f\})$, which maps bound variables to booleans. The boolean indicates whether the variable is going to be bound to a value, which makes a contribution to the computation. Similarly, for every occurrence of a constructor, we need to know whether its ith argument makes a contribution to the computation. Hence, we introduce another global environment, $\mathcal{G} : Label * Int \to Bool$. Here the label argument to $\mathcal{G}$ indicates the textual position of the data constructor.

The set-based operational semantics is presented in Table 6.1. For simplicity, all atomic operators have been left out of the language. We assume that all bound variables are distinct.

$$\mathcal{E}\,,\,\mathcal{F}\,,\,\mathcal{G}\,,\,b\,\vdash\,M\,\rightsquigarrow\,\{-\} \tag{1}$$

$$\frac{\mathcal{E}\,,\,\mathcal{F}\,,\,\mathcal{G}\,,\,t\,\vdash\,M_1\,\rightsquigarrow\,\{-\}}{\mathcal{E}\,,\,\mathcal{F}\,,\,\mathcal{G}\,,\,t\,\vdash\,l\,:\,M_1M_2\,\rightsquigarrow\,\{-\}} \tag{2}$$

$$\frac{\mathcal{E}\,,\,\mathcal{F}\,,\,\mathcal{G}\,,\,t\,\vdash\,M_1\,\rightsquigarrow\,\{-\}}{\mathcal{E}\,,\,\mathcal{F}\,,\,\mathcal{G}\,,\,t\,\vdash\,l\,:\,\text{case}(M_1\,,\,\text{C}(x_1\,,\,\ldots\,,\,x_n)\Rightarrow M_2\,,\,y\Rightarrow M_3)\,\rightsquigarrow\,\{-\}} \tag{3}$$

$$\mathcal{E}\,,\,\mathcal{F}\,,\,\mathcal{G}\,,\,t\,\vdash\,l\,:\,x\,\rightsquigarrow\,\mathcal{E}(x) \tag{4}$$

$$\mathcal{E}\,,\,\mathcal{F}\,,\,\mathcal{G}\,,\,t\,\vdash\,l\,:\,\lambda x\,M\,\rightsquigarrow\,\{\lambda x\,M\} \tag{5}$$

$$\frac{\mathcal{E}\,,\,\mathcal{F}\,,\,\mathcal{G}\,,\,t\,\vdash\,M_2\,\rightsquigarrow\,V}{\mathcal{E}\,,\,\mathcal{F}\,,\,\mathcal{G}\,,\,t\,\vdash\,l\,:\,\text{letrec}\,f(x)=M_1\,\text{in}\,M_2\,\rightsquigarrow\,V} \tag{6}$$

$$\frac{\begin{array}{c}\mathcal{E}\,,\,\mathcal{F}\,,\,\mathcal{G}\,,\,t\,\vdash\,M_1\,\rightsquigarrow\,V_1\\\mathcal{E}\,,\,\mathcal{F}\,,\,\mathcal{G}\,,\,\mathcal{F}(x)\,\vdash\,M_2\,\rightsquigarrow\,V_2\\\mathcal{E}\,,\,\mathcal{F}\,,\,\mathcal{G}\,,\,t\,\vdash\,M\,\rightsquigarrow\,V_3\end{array}}{\mathcal{E}\,,\,\mathcal{F}\,,\,\mathcal{G}\,,\,t\,\vdash\,l\,:\,M_1M_2\,\rightsquigarrow\,V_3} \tag{7}$$
$$\text{where}\,\lambda x.\,M\,\in\,V_1$$

$$\frac{\mathcal{E}\,,\,\mathcal{F}\,,\,\mathcal{G}\,,\,\mathcal{G}[l\,,\,i]\,\vdash\,M_i\,\rightsquigarrow\,V_i\,,\,i=1\ldots n}{\mathcal{E}\,,\,\mathcal{F}\,,\,\mathcal{G}\,,\,t\,\vdash\,l\,:\,\text{C}(M_1\,,\,\ldots\,,\,M_n)\,\rightsquigarrow\,C[l\,,\,(V_1\,,\,\ldots\,,\,V_n)]} \tag{8}$$

$$\frac{\mathcal{E}\,,\,\mathcal{F}\,,\,\mathcal{G}\,,\,t\,\vdash\,M_1\,\rightsquigarrow\,V'\quad\mathcal{E}\,,\,\mathcal{F}\,,\,\mathcal{G}\,,\,t\,\vdash\,M_2\,\rightsquigarrow\,V''}{\mathcal{E}\,,\,\mathcal{F}\,,\,\mathcal{G}\,,\,t\,\vdash\,l\,:\,\text{case}(M_1\,,\,\text{C}(x_1\,,\,\ldots\,,\,x_n)\Rightarrow M_2\,,\,y\Rightarrow M_3)\,\rightsquigarrow\,V''} \tag{9}$$
$$\text{where}\,\exists v\in V'\,s.t.\,v\equiv C[l_0\,,\,(\ldots)]$$

$$\frac{\mathcal{E}\,,\,\mathcal{F}\,,\,\mathcal{G}\,,\,t\,\vdash\,M_1\,\rightsquigarrow\,V'\quad\mathcal{E}\,,\,\mathcal{F}\,,\,\mathcal{G}\,,\,t\,\vdash\,M_3\,\rightsquigarrow\,V''}{\mathcal{E}\,,\,\mathcal{F}\,,\,\mathcal{G}\,,\,t\,\vdash\,l\,:\,\text{case}(M_1\,,\,\text{C}(x_1\,,\,\ldots\,,\,x_n)\Rightarrow M_2\,,\,y\Rightarrow M_3)\,\rightsquigarrow\,V''} \tag{10}$$
$$\text{where}\,\exists v\in V'\,s.t.\,v\equiv C'[l_0\,,\,(\ldots)]\,\text{and}\,C'\not\equiv C.$$

Table 6.1: Set-Based Operational Semantics

Given a term M, the rules in Table 6.1 can be used to construct a proof of $\mathcal{E}, \mathcal{F}, \mathcal{G}, b \vdash M \rightsquigarrow V$, where

b is a boolean indicating whether the value computed at this point contributes to the computation. If $b = f$, i.e., the value to be computed makes no contribution, then it is not computed: only an instance of rule (1) applies,

$$\mathcal{E}, \mathcal{F}, \mathcal{G}, f \vdash M \rightsquigarrow \{-\}$$

V is the set of values returned as a result of the computation,

$$V \quad ::= \quad \{ v_1, \ldots, v_n \}$$

$$v \quad ::= \quad -$$

$$| \quad C[l, (v_1, \ldots, v_n)]$$

$$| \quad \lambda x \, M$$

For every value built by the application of a data constructor, we need to keep track of the textual location where it was constructed. Such values are denoted by, $C[l, (v_1, \ldots, v_n)]$, where l is the textual location where the constructor C is applied to a tuple of values. The expression $C[l, (V_1, \ldots, V_n)]$ denotes the set of values $\{C[l, (v_1, \ldots, v_n) \mid v_i \in V_i\}$.

Note that the semantics is non-deterministic. This is because of the non-deterministic choice which needs to be made in rules (7), (9) & (10) and because of the fact that the rule,

$$\mathcal{E}, \mathcal{F}, \mathcal{G}, t \vdash M \rightsquigarrow \{-\}$$

can be used anywhere in the proof.

We now characterise the environments $(\mathcal{E}, \mathcal{F}, \mathcal{G})$ which provide a sound approximation to the value computed by the standard semantics.

Definition: $(\mathcal{E}, \mathcal{F}, \mathcal{G})$ is **safe** wrt a closed term M_0, if every derivation of the form $\mathcal{E}, \mathcal{F}, \mathcal{G}, t \vdash M_0 \rightsquigarrow V$ satisfies the following conditions:

A. In every use of Rule (4), $\mathcal{F}(x) = t$.

B. In every use of Rule (7), $V_2 \subseteq \mathcal{E}(x)$.

C. In every use of Rule (9), if $C[l_0, (v_1, \ldots, v_n)] \in V'$ then $\forall 1 \le i \le n$ if $\mathcal{F}(x_i)$ then $v_i \in \mathcal{E}(x_i)$ and $\mathcal{G}[l_0, i]$.

D. In every use of Rule (10), if $v \in V'$ and $v \ne C[\ldots]$ then $\mathcal{F}(y)$ implies $v \in \mathcal{E}(y)$.

E. If M_0 contains the term l : letrec $f(x) = M_1$ in M_2 then $\mathcal{F}(f)$ implies $\mathcal{E}(f) = \{\lambda x. M_1\}$.

F. If $C[l_0, \ldots] \in \mathcal{E}(x)$ then $l_0 : C(\ldots)$ is a subterm of M_0.

In Table 6.1, the rules (1), $(2)\,\&\,(3)$ are referred to as *empty* rules. In the absence of such rules, an environment may vacuously satisfy the safety conditions because, under the environment, there may be no terminating computation, hence, no complete proofs. The empty rules are needed to handle an important weakness of natural semantics: the inability to model a finite number of steps in a non-terminating computation. In the presence of empty rules a 'partial' proof, constituting a finite number of steps in a non-terminating computation, can be completed to form a legal proof.

The set-based semantics presented by Heintze in [30] does not have rules which are counterparts to our empty rules. This is a very serious lapse. Theorems explicitly stated by Heintze [30, page 311], e.g. Soundness and Minimality, are in fact *invalid*. The following program is a counter-example to his soundness theorem:

$$(\lambda x \ \text{case}(x, 0 \Rightarrow 2, y \Rightarrow \Omega)) \ 0$$

Note Ω is a nonterminating program. This is a program whose evaluation under the standard interpreter returns the value 2. Without the empty rules, an environment $\mathcal{E} \equiv [x \mapsto \{1\}]$ is vacuously safe, since it has no terminating computation associated with it. Adding our empty rules, with $\{-\}$ replaced by the empty set, $\emptyset$, restores soundness to his set-based semantics.

Lemma 6.3.1 (Minimality)
If $(\mathcal{E}_1, \mathcal{F}_1, \mathcal{G}_1)$ and $(\mathcal{E}_2, \mathcal{F}_2, \mathcal{G}_2)$ are safe wrt a closed term M_0, then so is $(\mathcal{E}_1 \cap \mathcal{E}_2, \mathcal{F}_1 \wedge \mathcal{F}_2, \mathcal{G}_1 \wedge \mathcal{G}_2)$.

If $\mathcal{E}_1 \cap \mathcal{E}_2 \,,\, \mathcal{F}_1 \wedge \mathcal{F}_2 \,,\, \mathcal{G}_1 \wedge \mathcal{G}_2 \,,\, t \vdash M \rightsquigarrow V \quad then$

$\exists V_1 \,,\, V_2. \quad \mathcal{E}_i \,,\, \mathcal{F}_i \,,\, \mathcal{G}_i \,,\, t \vdash M \rightsquigarrow V_i \quad where \quad V \subseteq V_i.$

Proof: Safety conditions E & F are dependent solely on the term M and are independent of computations associated with a given static environment. Hence, they are immediately valid in the set-based environment $(\mathcal{E}_1 \cap \mathcal{E}_2 \,,\, \mathcal{F}_1 \wedge \mathcal{F}_2 \,,\, \mathcal{G}_1 \wedge \mathcal{G}_2)$.

Given a proof tree of a computation, based on the semantics presented in Table 6.1, it is to be noticed that any time the boolean parameter, to the left of the $\vdash$, is false the first empty rule is used.

Given a proof tree for $\mathcal{E}_1 \cap \mathcal{E}_2 \,,\, \mathcal{F}_1 \wedge \mathcal{F}_2 \,,\, \mathcal{G}_1 \wedge \mathcal{G}_2 \,,\, t \vdash M \rightsquigarrow V$, but for the boolean parameter to the left of $\vdash$, identical safe proof-trees can be constructed for the set-based environments $\mathcal{E}_i \,,\, \mathcal{F}_i \,,\, \mathcal{G}_i$ returning values V_i, st $V \subseteq V_i$.

<div align="right">□</div>

Corollary 1 *Given a closed term M, there exists a minimum set-based environment $(\mathcal{E}_m \,,\, \mathcal{F}_m \,,\, \mathcal{G}_m)$ that is safe wrt it.*

The following two lemmas are used in the proof of the Soundness Theorem for our set-based semantics.

Lemma 6.3.2 *For any safe set-based environment $(\mathcal{E} \,,\, \mathcal{F} \,,\, \mathcal{G})$ wrt M_0, if there is a computation $\mathcal{E} \,,\, \mathcal{F} \,,\, \mathcal{G} \,,\, t \vdash M_0 \rightsquigarrow V_0$ which contains $\mathcal{E} \,,\, \mathcal{F} \,,\, \mathcal{G} \,,\, t \vdash M \rightsquigarrow \{-\}$ as a sub-proof then, for any valid computation $\mathcal{E} \,,\, \mathcal{F} \,,\, \mathcal{G} \,,\, t \vdash M \rightsquigarrow V$, there is a valid computation, $\mathcal{E} \,,\, \mathcal{F} \,,\, \mathcal{G} \,,\, t \vdash M_0 \rightsquigarrow V_0'$, which contains $\mathcal{E} \,,\, \mathcal{F} \,,\, \mathcal{G} \,,\, t \vdash M \rightsquigarrow V$ as a sub-proof.*

The above lemma can be easily proved by induction. This is because the set-based semantics does not create any bindings: evaluation proceeds under a global environment. It is the only in the rules for function application and case expressions that the value returned by the computation is significant: here we need to assume the type correctness of the program and the set-based environment.

Because of the presence of empty rules, the soundness theorem for safe environments can no longer be stated in terms of set-theoretic containment. Instead, the value, computed by the standard semantics, is proven to be related to the set of values returned by the set-based semantics by a relation $\epsilon_{(\mathcal{E},\mathcal{F},\mathcal{G})}$.

Definition: The **relation** $\epsilon_{(\mathcal{E},\mathcal{F},\mathcal{G})}$ is a relation between a value computed by the standard semantics and a set of values computed by the set-based semantics.

- $C(v_1, \ldots v_n) \, \epsilon_{(\mathcal{E},\mathcal{F},\mathcal{G})} \, V$ if $\exists C[l, (v'_1, \ldots v'_n)] \in V$,

  st $\forall i$ if $\mathcal{G}[l, i]$ then $v_i \, \epsilon_{(\mathcal{E},\mathcal{F},\mathcal{G})} \, \{v'_i\}$.

- $\langle E, \lambda x \, M \rangle \, \epsilon_{(\mathcal{E},\mathcal{F},\mathcal{G})} \, V$ if $(\lambda x \, M) \in V$ and $E \, \epsilon_{(\mathcal{E},\mathcal{F},\mathcal{G})} \, \mathcal{E}$.

- $\langle E, f, \lambda x \, M \rangle \, \epsilon_{(\mathcal{E},\mathcal{F},\mathcal{G})} \, V$ if $(\lambda x \, M) \in V$,

  $\mathcal{E}(f) = \{\lambda x \, M\}$ and $E \, \epsilon_{(\mathcal{E},\mathcal{F},\mathcal{G})} \, \mathcal{E}$.

- $E \, \epsilon_{(\mathcal{E},\mathcal{F},\mathcal{G})} \, \mathcal{E}$ if for each x, if $\mathcal{F}(x)$ then $E(x) \, \epsilon_{(\mathcal{E},\mathcal{F},\mathcal{G})} \, \mathcal{E}(x)$.

Given a proof tree $\mathcal{P}$ representing a computation in the set-based semantics, let us delete the boolean tags to the left of all occurrences of $\vdash$ to obtain a tree structure $\mathcal{T}$. Given the tree structure $\mathcal{T}$, we can reintroduce the boolean tags with their original values using a simple set of rules, obvious from the rule schemas for the set-based semantics, to obtain the proof tree $\mathcal{P}$. We are now going to use such rules to introduce a boolean tag to the left of $\vdash$ in a proof in the standard semantics.

Definition: Given a subterm M of a closed term M_0, and a safe set-based environment $(\mathcal{E},\mathcal{F},\mathcal{G})$ wrt M_0, the **boolean-annotated proof** of $E \vdash M \rightarrow v$ wrt $(\mathcal{E},\mathcal{F},\mathcal{G})$, $E, t \vdash M \rightarrow v$, is constructed as follows:

- Introduce the boolean tag t to the left of $\vdash$, at the root of the proof tree.

- Propagate the boolean tag towards the leaves of the proof-tree, in a manner similar to the technique used in the set-based operational semantics.

- The instant the boolean tag becomes f, the proof tree, in the set-based semantics, reaches its leaf. In the case of the annotated proof tree, we simply propagate the f tag all the way to the leaves.

Theorem 6.3.1 (Soundness) *If* $(\mathcal{E}, \mathcal{F}, \mathcal{G})$ *is safe wrt a closed term* M_0 *and* $\vdash M_0 \rightarrow v_0$, *then* $\exists V_0 \quad \mathcal{E}, \mathcal{F}, \mathcal{G}, t \vdash M_0 \rightsquigarrow V_0$, *st* $v_0 \, \epsilon_{(\mathcal{E},\mathcal{F},\mathcal{G})} \, V_0$.

Proof: Because of the presence of a boolean tag to the left of $\vdash$, in the set-based semantics and the fact that safety conditions apply only to complete proofs of M_0, the proof of soundness is not a simple structural induction on the proof of evaluation, based on the standard semantics. The induction hypothesis needs to be more elaborate. The key set of conditions to be used in the proof of the Soundness Theorem are those arising from the fact that $(\mathcal{E}, \mathcal{F}, \mathcal{G})$ is safe wrt the term M_0. The presence of the boolean tag to the left of $\vdash$, in the set-based semantics, prevents us from asserting the fact that if $(\mathcal{E}, \mathcal{F}, \mathcal{G})$ is safe wrt a closed term term M_0 then $(\mathcal{E}, \mathcal{F}, \mathcal{G})$ is also safe wrt any subterm M of M_0.

The induction hypothesis:

If   (a) $E, t \vdash M \rightarrow v$ is a boolean-annotated proof of $E \vdash M \rightarrow v$ wrt $(\mathcal{E}, \mathcal{F}, \mathcal{G})$,

    (b) $E \, \epsilon_{(\mathcal{E}, \mathcal{F}, \mathcal{G})} \, \mathcal{E}$,

    (c)   There is a proof $\mathcal{E}, \mathcal{F}, \mathcal{G}, t \vdash M_0 \rightsquigarrow V_0$ which contains

        $\mathcal{E}, \mathcal{F}, \mathcal{G}, t \vdash M \rightsquigarrow \{-\}$ as a subproof.

Then there exists a proof, $\mathcal{E}, \mathcal{F}, \mathcal{G}, t \vdash M \rightsquigarrow V$, such that $v \, \epsilon_{(\mathcal{E}, \mathcal{F}, \mathcal{G})} \, V$.

Condition (c) is required because safety conditions apply only to complete proofs of M_0. The cases of the induction are discussed below:

Rule (Var): Let $M \equiv x$. This is trivially true if $\mathcal{F}(x)$ is true. Since, there is a computation $\mathcal{E}, \mathcal{F}, \mathcal{G}, t \vdash M_0 \rightsquigarrow V_0$ which contains $\mathcal{E}, \mathcal{F}, \mathcal{G}, t \vdash x \rightsquigarrow \{-\}$ as a subproof, by Lemma 6.3.2, there is a computation $\mathcal{E}, \mathcal{F}, \mathcal{G}, t \vdash M_0 \rightsquigarrow V_0$ which contains $\mathcal{E}, \mathcal{F}, \mathcal{G}, t \vdash x \rightsquigarrow \mathcal{E}(x)$ as a sub-proof. The falsity of $\mathcal{F}(x)$ would then show that $\mathcal{E}, \mathcal{F}, \mathcal{G}$ is not safe for M_0. This is a contradiction.

Rule (App): Let $M \equiv M_1 M_2$ and $M_1 \rightarrow \langle E_1, \lambda x \, M' \rangle$. By assumption, there is a computation $\mathcal{E}, \mathcal{F}, \mathcal{G}, t \vdash M_0 \rightsquigarrow V_0$ which contains $\mathcal{E}, \mathcal{F}, \mathcal{G}, t \vdash M_1 M_2 \rightsquigarrow \{-\}$ as a subproof.

The subproof $\mathcal{E}, \mathcal{F}, \mathcal{G}, t \vdash M_1 M_2 \rightsquigarrow \{-\}$ can be replaced by the subproof:

$$\frac{\mathcal{E}, \mathcal{F}, \mathcal{G}, t \vdash M_1 \rightsquigarrow \{-\}}{\mathcal{E}, \mathcal{F}, \mathcal{G}, t \vdash M_1 M_2 \rightsquigarrow \{-\}}$$

Hence, by induction, $\langle E_1, \lambda x \, M' \rangle \, \epsilon_{(\mathcal{E}, \mathcal{F}, \mathcal{G})} \, V_1$, i.e $(\lambda x \, M') \in V_1$ and $E_1 \, \epsilon_{(\mathcal{E}, \mathcal{F}, \mathcal{G})} \, \mathcal{E}$.

The subproof $\mathcal{E}$, $\mathcal{F}$, $\mathcal{G}$, $t \vdash M_1 M_2 \rightsquigarrow \{-\}$ can be replaced by the subproof:

$$\mathcal{E} , \mathcal{F} , \mathcal{G} , t \vdash M_1 \rightsquigarrow V_1$$
$$\mathcal{E} , \mathcal{F} , \mathcal{G} , \mathcal{F}(x) \vdash M_2 \rightsquigarrow \{-\}$$
$$\frac{\mathcal{E} , \mathcal{F} , \mathcal{G} , t \vdash M' \rightsquigarrow \{-\}}{\mathcal{E} , \mathcal{F} , \mathcal{G} , t \vdash M_1 M_2 \rightsquigarrow \{-\}}$$

Hence, if $\mathcal{F}(x)$ then, by induction, $v_2 \, \epsilon_{(\mathcal{E},\mathcal{F},\mathcal{G})} \, V_2$. In the above subproof, if $M_2 \rightsquigarrow \{-\}$ is replaced by a proof of $M_2 \rightsquigarrow V_2$ then by the safety property of $(\mathcal{E} , \mathcal{F} , \mathcal{G})$ wrt M_0, we have $V_2 \subseteq \mathcal{E}(x)$, hence, $E_1[x \mapsto v_2] \, \epsilon_{(\mathcal{E},\mathcal{F},\mathcal{G})} \, \mathcal{E}$. Hence, if $M \equiv M_1 M_2 \rightarrow v$ then $v \, \epsilon_{(\mathcal{E},\mathcal{F},\mathcal{G})} \, V$.

If $\mathcal{F}(x)$ is false then $E_1[x \mapsto v_2] \, \epsilon_{(\mathcal{E},\mathcal{F},\mathcal{G})} \, \mathcal{E}$ for any value $v2$. Hence, the proof follows trivially.

Rule (Cons): Let $M \equiv C(M_1 , \ldots , M_n)$. By assumption, there is a computation $\mathcal{E} , \mathcal{F} , \mathcal{G} , t \vdash M_0 \rightsquigarrow V_0$ which contains $\mathcal{E} , \mathcal{F} , \mathcal{G} , t \vdash C(M_1 , \ldots , M_n) \rightsquigarrow \{-\}$ as a subproof. By Lemma 6.3.2, this implies that there is a computation $\mathcal{E} , \mathcal{F} , \mathcal{G} , t \vdash M_0 \rightsquigarrow V_0'$ which contains $\mathcal{E} , \mathcal{F} , \mathcal{G} , t \vdash M_i \rightsquigarrow \{-\}$ as a subproof. Hence, for every i such that $1 \leq i \leq n$, if $\mathcal{G}[l, i] = t$ then, by induction, $v_i \, \epsilon_{(\mathcal{E},\mathcal{F},\mathcal{G})} \, V_i$.

Hence, $C(v_1 , \ldots , v_n) \, \epsilon_{(\mathcal{E},\mathcal{F},\mathcal{G})} \, C[l , (V_1 , \ldots , V_n)]$.

Rule (Case1): Let $M \equiv \text{case}(M_1 , \text{C}(x_1 , \ldots , x_n) \Rightarrow M_2 , y \Rightarrow M_3)$. If $M_1 \rightarrow C(v_1 , \ldots , v_n)$ then, by induction, $C(v_1 , \ldots , v_n) \, \epsilon_{(\mathcal{E},\mathcal{F},\mathcal{G})} \, V'$, i.e. $\exists \, C[l' , (v_1' , \ldots , v_n')] \in V$, for some l', st if $\mathcal{G}[l', i] = t$ then $v_i \, \epsilon_{(\mathcal{E},\mathcal{F},\mathcal{G})} \, \{v_i'\}$.

By assumption, $\mathcal{E} , \mathcal{F} , \mathcal{G} , t \vdash M \rightsquigarrow \{-\}$ is a subproof of some computation $\mathcal{E} , \mathcal{F} , \mathcal{G} , t \vdash M_0 \rightsquigarrow V_0$. Hence, the following subcomputation,

$$\mathcal{E} , \mathcal{F} , \mathcal{G} , t \vdash M_1 \rightsquigarrow V'$$
$$\mathcal{E} , \mathcal{F} , \mathcal{G} , t \vdash M_2 \rightsquigarrow \{-\}$$
$$\overline{\mathcal{E} , \mathcal{F} , \mathcal{G} , t \vdash \text{case}(M_1 , \text{C}(x_1 , \ldots , x_n) \Rightarrow M_2 , y \Rightarrow M_3) \rightsquigarrow \{-\}}$$

can be made to be a part of the proof of $\mathcal{E} , \mathcal{F} , \mathcal{G} , t \vdash M_0 \rightsquigarrow V_0$. The safety property of $(\mathcal{E} , \mathcal{F} , \mathcal{G})$ wrt M_0 implies that if $\mathcal{F}(x_i)$ then $v_i \in \mathcal{E}(x_i)$ and $\mathcal{G}[l', i] = t$. Hence, $E[x_1 \mapsto v_1 , \ldots , x_n \mapsto v_n] \, \epsilon_{(\mathcal{E},\mathcal{F},\mathcal{G})} \, \mathcal{E}$. We can now apply induction, to obtain $v'' \, \epsilon_{(\mathcal{E},\mathcal{F},\mathcal{G})} \, V''$.

Rule (Case2): If $M_1 \to C'(v_1, \ldots, v_n)$ then, by induction,

$C'(v_1, \ldots, v_n) \,\epsilon_{(\mathcal{E},\mathcal{F},\mathcal{G})} \, V'$, i.e. $C'[l', (v'_1, \ldots, v'_n)] \in V$, for some l'.

By assumption, $\mathcal{E}, \mathcal{F}, \mathcal{G}, t \vdash M \rightsquigarrow \{-\}$ is a subproof of some computation $\mathcal{E}, \mathcal{F}, \mathcal{G}, t \vdash M_0 \rightsquigarrow V_0$. Hence, the following subcomputation,

$$\frac{\begin{array}{c} \mathcal{E}, \mathcal{F}, \mathcal{G}, t \vdash M_1 \rightsquigarrow V' \\ \mathcal{E}, \mathcal{F}, \mathcal{G}, t \vdash M_2 \rightsquigarrow \{-\} \end{array}}{\mathcal{E}, \mathcal{F}, \mathcal{G}, t \vdash \text{case}(M_1, \text{C}(x_1, \ldots, x_n) \Rightarrow M_2, y \Rightarrow M_3) \rightsquigarrow \{-\}}$$

can be made to be a part of the proof of $\mathcal{E}, \mathcal{F}, \mathcal{G}, t \vdash M_0 \rightsquigarrow V_0$.

The safety property of $(\mathcal{E}, \mathcal{F}, \mathcal{G})$ wrt M_0 implies that if $\mathcal{F}(y)$ then $C'[l', (v_1, \ldots, v_n)] \in \mathcal{E}(y)$. Hence, $E[y \mapsto C'(v_1, \ldots, v_n)] \,\epsilon_{(\mathcal{E},\mathcal{F},\mathcal{G})} \, \mathcal{E}$. The result follows by induction.

$\square$

Definition: Given a closed term M_0 and a boolean-annotated proof tree P of the form, $t \vdash M_0 \to v_0$,

$$\textbf{slice}(M_0, P) = \{ l \mid E, t \vdash l : e \to v \text{ is a subproof of } P \}$$

Given a terminating computation and a safe set-based environment $(\mathcal{E}, \mathcal{F}, \mathcal{G})$, the Slicing Theorem(Theorem 6.3.2) shows how subterms which make no contribution to the computation can be specified. If the value returned by a program is a nullary constructor, then the deletion of subterms, which do not contribute to the computation, causes the same value to be returned. But if the value returned is not a nullary constructor then we need to define a formal relation $\mathcal{R}$ to specify the value returned on the deletion of subterms which do not contribute to the computation. For this theorem, we will assume that a value built by the application of a data constructor, in the standard interpreter, also includes the label of the textual location where it was constructed.

Definition: $v \, \mathcal{R}_{(\mathcal{F},\mathcal{G},\mathcal{L}')} \, v'$ is a relation between two values computed by the standard semantics before and after the deletion of a set of subterms $\mathcal{L}'$.

- $C[l\,,\,(v_1\,,\,\ldots v_n)]\; \mathcal{R}_{(\mathcal{F},\mathcal{G},\mathcal{L'})}\; C[l\,,\,(v'_1\,,\,\ldots v'_n)]$
  if for every $\mathcal{G}[l\,,\,i] = t,\;\; v_i\; \mathcal{R}_{(\mathcal{F},\mathcal{G},\mathcal{L'})}\; v'_i.$

- $\langle E\,,\,\lambda x\,M\rangle\; \mathcal{R}_{(\mathcal{F},\mathcal{G},\mathcal{L'})}\; \langle E'\,,\,\lambda x\,M'\rangle$
  if $M' \equiv M[\mathcal{L'}/-]\;$ and $\; E\; \mathcal{R}_{(\mathcal{F},\mathcal{G},\mathcal{L'})}\; E'.$

- $\langle E\,,\,f\,,\,\lambda x\,M\rangle\; \mathcal{R}_{(\mathcal{F},\mathcal{G},\mathcal{L'})}\; \langle E'\,,\,f\,,\,\lambda x\,M'\rangle$
  if $M' \equiv M[\mathcal{L'}/-]\;$ and $\; E\; \mathcal{R}_{(\mathcal{F},\mathcal{G},\mathcal{L'})}\; E'.$

- $E\; \mathcal{R}_{(\mathcal{F},\mathcal{G},\mathcal{L'})}\; E'\;$ if $\;$ for each x, such that $\mathcal{F}(x) = t,\;\; E(x)\; \mathcal{R}_{(\mathcal{F},\mathcal{G},\mathcal{L'})}\; E'(x).$

Since, the standard natural semantics can only represent terminating computations we need a lemma to assert that deletion of subterms in a terminating purely functional program creates another terminating program. This is done by defining an order on values returned by a program:

- $-\;\leq\;v$

- $C(v_1\,,\,\ldots\,,\,\ldots)\;\leq\;C(v'_1\,,\,\ldots\,,\,l'_n)\;$ if $\forall i,\;v_i\;\leq v'_i.$

- $\langle E\,,\,\lambda x\,M\rangle\;\leq\;\langle E'\,,\,\lambda x\,M'\rangle\; if\;(E\;\leq\;E')\;$ and $\;M'[\mathcal{L}/-] = M$ for some set of subterms $\mathcal{L}.$

- $E\;\leq\;E'\;$ if $\;E(x) \leq E'(x)$, for every $x.$

Lemma 6.3.3 *If $\;E\;\vdash\;M\;\to\;v\;$ then for $E \geq E',\;\;E'\;\vdash\;M[\mathcal{L'}/-]\;\to\;v'$, where $v \geq v'.$*

This lemma can be easily proved by induction on the height of the proof tree/computation.

Theorem 6.3.2 (Static Slicing Theorem)

Let $\;(\mathcal{E},\mathcal{F},\mathcal{G})\;\;$ be a safe set-based environment for a closed term $M_0.$

Let $\;\;\vdash\;M_0\;\to\;v_0\;\;$ be the terminating computation under the standard interpreter.

Let $\;P \equiv (t \vdash M_0\;\to\;v_0)\;\;$ be the boolean-annotated proof of $\;\vdash\;M_0\;\to\;v_0\;\;$ wrt $\;(\mathcal{E},\mathcal{F},\mathcal{G}).$

Let $\;\mathcal{L}_0 = slice(M_0\,,\,P).$

If $\;\mathcal{L'}_0 = \Lambda(M_0) - \mathcal{L}_0\;\;$ then $\;\;\vdash\;M_0[\mathcal{L'}_0/-]\;\to\;v'_0,$
where $v_0\;\mathcal{R}_{(\mathcal{F},\mathcal{G},\mathcal{L'})}\;v'_0\;\;$ and $\;\Lambda(M_0)\;$ is the set of labels contained in $M_0.$

Proof: The proof is given by induction on the structure of the proof tree and applications of the Soundness Theorem.

We will assume here that the standard interpreter also tags values, built by constructor applications, with labels of the site in which they were created.

Induction Hypothesis:

Given a subproof $E, t \vdash M \rightarrow v$, if $E \mathcal{R}_{(\mathcal{F},\mathcal{G},\mathcal{L}')} E'$ then there exists a proof $E', t \vdash M[\mathcal{L}'/-] \rightarrow v'$, where $v \mathcal{R}_{(\mathcal{F},\mathcal{G},\mathcal{L}')} v'$.

Rule(Var): Let $M \equiv x$ be a subterm of M_0. Given a proof $t \vdash M_0 \rightarrow v_0$ and a set-based environment, $(\mathcal{E}, \mathcal{F}, \mathcal{G})$, safe wrt M_0, we can, by the Soundness theorem, develop a similar set-based proof for M_0. By the safety conditions on this set-based proof, we have that $\mathcal{F}(x)$ is true. Hence, by assumption, $E(x) \mathcal{R}_{(\mathcal{F},\mathcal{G},\mathcal{L}')} E'(x)$.

Rule(App): Let $M \equiv l : (l_1 : e_1)(l_2 : e_2)$. Let $E, t \vdash l_1 : e_1 \rightarrow \langle E_1, \lambda x M_1 \rangle$. By induction, $E' \vdash (l_1 : e_1)[\mathcal{L}'/-] \rightarrow \langle E'_1, \lambda x M_1[\mathcal{L}'/-] \rangle$, where $E_1 \mathcal{R}_{(\mathcal{F},\mathcal{G},\mathcal{L}')} E'_1$.

If $\mathcal{F}(x)$ is true and $E, t \vdash l_2 : e_2 \rightarrow v_2$ then, by induction, $E' \vdash (l_2 : e_2)[\mathcal{L}'/-] \rightarrow v'_2$, where $v_2 \mathcal{R}_{(\mathcal{F},\mathcal{G},\mathcal{L}')} v'_2$.

Hence, $E[x \mapsto v_2] \mathcal{R}_{(\mathcal{F},\mathcal{G},\mathcal{L}')} E'[x \mapsto v'_2]$. The result now follows, by induction.

If $\mathcal{F}(x)$ is false then if the computation of $(l_2 : e_2)[\mathcal{L}'/-]$ terminates the result follows immediately by induction.

Termination of the computation of $(l_2 : e_2)[\mathcal{L}'/-]$ is a non-issue if $l_2 \in \mathcal{L}$. Otherwise, we apply Lemma 6.3.3 to assert the termination of the computation of $(l_2 : e_2)[\mathcal{L}'/-]$.

Rule(Cons): If $\mathcal{G}[l, i]$ is false then $l_i \in \mathcal{L}$, i.e. the l_i^{th} subterm has been deleted. The proof can now be completed by induction.

Rule(Case1): Let $M \equiv l : \text{case}(l_1 : e_1, \text{C}(x_1, \ldots, x_n) \Rightarrow l_2 : e_2, y \Rightarrow l_3 : e_3)$. Let $(l_1 : e_1)$ evaluate to $C[l_0, (v_1, \ldots, v_n)]$. Let $(l_1 \ e_1)[\mathcal{L}'/-]$ evaluate to $C[l_0, (v'_1, \ldots, v'_n)]$. For any i such that, $\mathcal{G}(l_0, i) = t$, we have $v_i \mathcal{R}_{(\mathcal{F},\mathcal{G},\mathcal{L}')} v'_i$. Since $(\mathcal{E}, \mathcal{F}, \mathcal{G})$ is safe wrt M_0, if $\mathcal{F}(x_i)$ then $\mathcal{G}(l_0, i)$. For any i such that, $\mathcal{G}[l\_0, i] = f$, we apply Lemma 6.3.3 to assert termination of computation. Hence, $E[x_1 \mapsto v_1, \ldots x_n \mapsto v_n] \mathcal{R}_{(\mathcal{F},\mathcal{G},\mathcal{L}')} E'[x_1 \mapsto v'_1, \ldots x_n \mapsto v'_n]$. We can now apply the

induction hypothesis to obtain the result.

The induction for **Case 2** can be similarly completed.

$$\square$$

The Soundness Theorem and the Static Slicing Theorem together provide a declarative specification of a set of subterms which make no contribution to the computation. Let $(\mathcal{E}, \mathcal{F}, \mathcal{G})$ be a safe set-based environment wrt M_0. Let P be the boolean-annotated proof tree, wrt this set-based environment, of the computation of the standard interpreter. If a subterm l contributes to the computation then, by the Static Slicing Theorem, $l \in slice(M_0, P)$. The Soundness Theorem states that a proof tree in the standard semantics can be played out in the set-based semantics. Hence, the set $\mathcal{L}(\mathcal{E}, \mathcal{F}, \mathcal{G})$, defined below, contains $slice(M_0, P)$. Thus, the complement of $\mathcal{L}(\mathcal{E}, \mathcal{F}, \mathcal{G})$ is a subset of the subterms which make no contribution to the computation.

Definition: Given a safe set-based environment $(\mathcal{E}, \mathcal{F}, \mathcal{G})$ wrt a closed term M_0, the set $\mathcal{L}(\mathcal{E}, \mathcal{F}, \mathcal{G})$, contains the subterm l iff there exists a subproof $\mathcal{E}, \mathcal{F}, \mathcal{G}, t \vdash l : e \rightsquigarrow V$, of some proof $\mathcal{E}, \mathcal{F}, \mathcal{G}, t \vdash M_0 \rightsquigarrow V_0$.

It should be noticed that, by the proof of the Minimality Theorem, the function $\mathcal{L}$ is monotonic in its argument, i.e. if $(\mathcal{E}_1, \mathcal{F}_1, \mathcal{G}_1) \geq (\mathcal{E}_2, \mathcal{F}_2, \mathcal{G}_2)$ then $\mathcal{L}(\mathcal{E}_1, \mathcal{F}_1, \mathcal{G}_1) \supseteq \mathcal{L}(\mathcal{E}_2, \mathcal{F}_2, \mathcal{G}_2)$. Hence, we define the set-based approximation for dead code, $sba_{deadcode}$, using the minimum safe set-based environment, as follows,

Definition: $sba_{deadcode}(M_0) = \Lambda(M_0) - \mathcal{L}(\mathcal{E}_m, \mathcal{F}_m, \mathcal{G}_m)$ where, $(\mathcal{E}_m, \mathcal{F}_m, \mathcal{G}_m)$ is the minimum safe set-based environment wrt M_0.

6.4 Constraints

Given a terminating computation and a safe set-based environment, the Slicing Theorem presents a way of specifying subterms which make no contribution to the computation. In this section, we are going to present an algorithm for computing a safe set-based environment for a closed term.

The approach taken is similar to [30]:

64

- Define a language for expressing constraints.

- Develop a concept of a model for such constraints.

- Develop an algorithm for the computation of the minimum model of such constraints.

- Develop an algorithm to infer a collection of constraints from a given program.

- Relate the minimum model of the above constraints to the minimum safe set-based environment.

6.4.1   The Language of Constraints

The set-based semantics defined computation wrt global environments:

- $\mathcal{E} : Var \rightarrow \mathcal{P}(Val)$.

- $\mathcal{F} : Var \rightarrow Bool$.

- $\mathcal{G} : Label * Int \rightarrow Bool$.

The environments $\mathcal{E}$, $\mathcal{F}$, $\mathcal{G}$ are finite mappings. Ideally, we would like to have constraints, having free variables corresponding to the elements of the domains of these finite mappings, i.e. variables denoted by E_x, F_x & $G_{l,i}$, such that a solution to these constraints provides us with the environments we seek to compute.

The constraints constructed from a program have more free variables. The solution to the constraints returns environments with the following augmented domains:

- $\mathcal{E} : Var \cup Label \rightarrow \mathcal{P}(Val)$.

- $\mathcal{F} : Var \cup Label \rightarrow Bool$.

- $\mathcal{G} : Label * Int \rightarrow Bool$.

Hence, we have constraints with variables E_x, E_l, F_x, F_l & $G_{l,i}$. We will use the letter z to generically refer to both variables and labels. To avoid notational clutter, variables E_x & E_l will simply be referred to as x & l.

Constraints are partitioned into set constraints and boolean constraints. This partition has been made so that our constraint language is a modular extension of the constraint

language defined in [30]. Such a partition at a syntactic level, however, necessitates some overlap/recomputation at the semantic/simplification level.

Set constraints $\mathcal{S}$, contain free variables x & l which are to be mapped to a subset of $\mathcal{P}(\textit{Val})$. Boolean constraints $\mathcal{B}$, contain free variables F_z & $\mathcal{G}_{l,i}$ which are to be mapped to *Bool*. We will use the letter Y to generically refer to boolean variables F_z & $\mathcal{G}_{l,i}$.

The language of *atomic set expressions* is given by,

$$
\begin{aligned}
ae \quad ::= \quad & z \\
| \quad & \lambda x\, M \\
| \quad & C[l\,,\,(l_1\,,\,\ldots\,,\,l_n)]
\end{aligned}
$$

The language of *set constraints* is given by,

$$
\begin{aligned}
\mathcal{S} \quad ::= \quad & z \supseteq ae \\
| \quad & l \supseteq \mathrm{Apply}(l_1\,,\,l_2) \\
| \quad & l \supseteq \mathrm{Case}(l_1\,,\,C(x_1\,,\,\ldots\,,\,x_n) \Rightarrow l_2\,,\,y \Rightarrow l_3)
\end{aligned}
$$

The language of *boolean constraints* is given by,

$$
\begin{aligned}
\mathcal{B} \quad ::= \quad & F_z \\
| \quad & \mathcal{G}_{l,i} \\
| \quad & F_{z_1} \Rightarrow F_{z_2} \\
| \quad & F_x \Rightarrow \mathcal{G}_{l,i} \\
| \quad & F_{l_j} \wedge \mathcal{G}_{l,i} \Rightarrow F_{l_i} \\
| \quad & \mathrm{FApply}(l_1\,,\,l_2) \\
| \quad & \mathrm{FCase1}(l_1\,,\,C(x_1\,,\,\ldots\,,\,x_n) \Rightarrow l_2\,,\,y \Rightarrow l_3) \\
| \quad & \mathrm{FCase2}(l_1\,,\,C(x_1\,,\,\ldots\,,\,x_n) \Rightarrow l_2\,,\,y \Rightarrow l_3)
\end{aligned}
$$

A collection of constraints $(\mathcal{S}\,,\,\mathcal{B})$ has a model $(\mathcal{E},\mathcal{F},\mathcal{G})$, if every constraint, under this model, is satisfied. The denotation of atomic set expressions and atomic boolean expressions, under environments $(\mathcal{E},\mathcal{F},\mathcal{G})$, is given in Table 6.2. Note that, the denotation of Apply and Case set expressions is defined, under a given model, only if the stated side-conditions are satisfied under the model. If $\mathcal{F}(l_1) = \text{false}$ then both $[\![\mathrm{Apply}(l_1,\ldots)]\!]$ and $[\![\mathrm{Case}(l_1,\ldots)]\!]$ equal $\emptyset$.

$$[\![\, z \,]\!]_{(\mathcal{E},\mathcal{F},\mathcal{G})} \;=\; \mathcal{E}(z)$$

$$[\![\, \mathrm{C}[l\,,\,(l_1\,,\,\ldots\,,\,l_n)]\,]\!]_{(\mathcal{E},\mathcal{F},\mathcal{G})} \;=\; \{\; C[l\,,\,(v_1\,,\,\ldots\,,\,v_n)]\;|\;v_i \in \mathcal{E}(l_i)\;\}$$

$$[\![\,(\lambda x.\;l\,:\,e)\,]\!]_{(\mathcal{E},\mathcal{F},\mathcal{G})} \;=\; \{\,(\lambda x.\;l\,:\,e)\,\}$$

$$[\![\,\mathrm{Apply}(l_1\,,\,l_2)\,]\!]_{(\mathcal{E},\mathcal{F},\mathcal{G})} \;=\; \{\;v\;|\;(\lambda x.\;l_0\,:\,e_0) \in \mathcal{E}(l_1)\,,\;\mathcal{F}(l_1) = t \text{ and } v \in \mathcal{E}(l_0)\;\}$$
$$\text{provided, for any } (\lambda x.\;l_0\,:\,e_0) \in \mathcal{E}(l_1),\;\;\mathcal{F}(x) \wedge \mathcal{F}(l_1) \text{ implies } \mathcal{E}(x) \supseteq \mathcal{E}(l_2)$$

$$[\![\,\mathrm{Case}(l_1\,,\,\mathrm{C}(x_1\,,\,\ldots\,,\,x_n) \Rightarrow l_2\,,\,y \Rightarrow l_3)\,]\!]_{(\mathcal{E},\mathcal{F},\mathcal{G})} \;=\; S_1 \cup S_2$$
$$\text{where,}\quad S_1 \;=\; \{\;v\;|\;v \in \mathcal{E}(l_2),\;\text{if } C[\ldots] \in \mathcal{E}(l_1) \text{ and } \mathcal{F}(l_1) = t\;\}$$
$$\text{provided,}\quad \forall i\;\forall C[l_0\,,\,(v_1\,,\,\ldots\,,\,v_n)] \in \mathcal{E}(l_1),$$
$$\mathcal{F}(x_i) \wedge \mathcal{F}(l_1) \text{ implies } v_i \in \mathcal{E}(x_i)$$
$$S_2 \;=\; \{\;v\;|\;v \in \mathcal{E}(l_3),\;\text{if } C'[\ldots] \in \mathcal{E}(l_1) \text{ and } \mathcal{F}(l_1) = t\;\}$$
$$\text{provided,}\quad \text{for any } v \equiv C'[\ldots] \in \mathcal{E}(l_1),$$
$$\mathcal{F}(y) \wedge \mathcal{F}(l_1) \text{ implies } v \in \mathcal{E}(y)$$

$$[\![\,\mathrm{FApply}(l_1\,,\,l_2)\,]\!]_{(\mathcal{E},\mathcal{F},\mathcal{G})} \;=\; \mathcal{F}(l_1) \text{ and } \vee\{\;\mathcal{F}(x)\;|\;\lambda x.\;(l\,:\,e) \in \mathcal{E}(l_1)\;\} \text{ imply } \mathcal{F}(l_2)$$
$$\text{and}\;\;\mathcal{F}(l_1) \text{ implies } \wedge\{\;\mathcal{F}(l)\;|\;\lambda x\;(l\,:\,e) \in \mathcal{E}(l_1)\;\}$$

$$[\![\,\mathrm{FCase1}(l_1\,,\,\mathrm{C}(x_1\,,\,\ldots\,,\,x_n) \Rightarrow l_2\,,\,y \Rightarrow l_3)\,]\!]_{(\mathcal{E},\mathcal{F},\mathcal{G})} \;=$$
$$(C[l_0\,,\,\ldots] \in \mathcal{E}(l_1) \text{ and } \mathcal{F}(l_1)) \text{implies}\;\;\mathcal{F}(l_2) \text{ and } \mathcal{F}(x_i) \Rightarrow \mathcal{G}[l_0\,,\,i]$$

$$[\![\,\mathrm{FCase2}(l_1\,,\,\mathrm{C}(x_1\,,\,\ldots\,,\,x_n) \Rightarrow l_2\,,\,y \Rightarrow l_3)\,]\!]_{(\mathcal{E},\mathcal{F},\mathcal{G})} \;=$$
$$(C'[\ldots] \in \mathcal{E}(l_1) \text{ and } \mathcal{F}(l_1))\;\;\text{implies}\;\;\mathcal{F}(l_3).$$

Table 6.2 : Denotation of Atomic Set-Based Expressions

input a collection $(\mathcal{S}, \mathcal{B})$ of set and boolean constraints.

repeat

  /* *Simplification of Boolean Constraints* */

  if $\{ Y \Rightarrow F_l , \ Y \} \subseteq \mathcal{B}$
    then add F_l to $\mathcal{B}$.

  if $\{ Y_1 \wedge Y_2 \Rightarrow F_l , \ Y_1 , \ Y_2 \} \subseteq \mathcal{B}$
    then add F_l to $\mathcal{B}$.

  if $\{ \mathrm{FApply}(l_1 , l_2) , \ F_{l_1} \} \subseteq \mathcal{B}$ and $(l_1 \supseteq \lambda x \, (l_0 : e_0)) \in \mathcal{S}$
    then add $\{ F_{l_0} , \ F_x \Rightarrow F_{l_2} \}$ to $\mathcal{B}$.

  if $\{ \mathrm{FCase1}(l_1, \mathrm{C}(x_1 , \ldots , x_n) \Rightarrow l_2, y \Rightarrow l_3), F_{l_1} \} \subseteq \mathcal{B}$ & $(l_1 \supseteq C[l_0 , \ldots]) \in \mathcal{S}$
    then add F_{l_2} to $\mathcal{B}$.
        for every i, add $F_{x_i} \Rightarrow G_{l_0,i}$ to $\mathcal{B}$.

  if $\{ \mathrm{FCase2}(l_1, \mathrm{C}(x_1 , \ldots , x_n) \Rightarrow l_2 , y \Rightarrow l_3), F_{l_1} \} \subseteq \mathcal{B}$ & $(l_1 \supseteq C'[l_0 , \ldots]) \in \mathcal{S}$
    then add F_{l_3} to $\mathcal{B}$.

  /* *Simplification of Set Constraints* */

  if $\{ z \supseteq l_1 , \ l_1 \supseteq ae \} \subseteq \mathcal{S}$, where ae is atomic and not a set variable,
    then add $z \supseteq ae$ to $\mathcal{S}$.

  if $\{ l \supseteq \mathrm{Apply}(l_1 , l_2) , \ l_1 \supseteq \lambda x \, (l_0 : e_0) \} \subseteq \mathcal{S}$ and $F_{l_1} \in \mathcal{B}$
    then add $l \supseteq l_0$ to $\mathcal{S}$.
        if $F_x \in \mathcal{B}$ then add $x \supseteq l_2$ to S.

  if $(l \supseteq \mathrm{Case}(l_1' , \mathrm{C}(x_1 , \ldots , x_n) \Rightarrow l_2' , y \Rightarrow l_3')) \in \mathcal{S}$ and $F_{l_1'} \in \mathcal{B}$
    then if $(l_1' \supseteq C[l_0 , (l_1 , \ldots , l_n)]) \in \mathcal{S}$
      then add $l \supseteq l_2'$ to $\mathcal{S}$.
          for every i, if $F_{x_i} \in \mathcal{B}$ then add $x_i \supseteq l_i$ to $\mathcal{S}$.
    if $(l_1' \supseteq (X \equiv C'[\ldots])) \in \mathcal{S}$ then add $l \supseteq l_3'$ to $\mathcal{S}$.
          if $F_y \in \mathcal{B}$ then add $y \supseteq X$ to $\mathcal{S}$.

until No changes in $(\mathcal{S}, \mathcal{B})$.

output $explicit(\mathcal{S}, \mathcal{B})$.

Table 6.3 : Set Constraint Simplification Algorithm, *Simplify*

68

A minimum model for a collection of constraints is computed by a process of simplification of the constraints. The algorithm, *Simplify*, is presented in Table 6.3.

Theorem 6.4.1 *If* $(\mathcal{E}, \mathcal{F}, \mathcal{G})$ *is a model of a set of constraints* $(\mathcal{S}, \mathcal{B})$ *then* $(\mathcal{E}, \mathcal{F}, \mathcal{G})$ *is also a model of* $(\mathcal{S}', \mathcal{B}')$, *the set of constraints obtained after applying a single step of the simplification algorithm.*

Proof: The proof is by cases, over the new constraints introduced by the simplification algorithm.

$\square$

Definition: A **set constraint** is in **explicit form**, if it has the form $z \supseteq ae$ where ae is an atomic expression that is not a variable.

A **boolean constraint** is in **explicit form**, if it is a variable.

Theorem 6.4.2 *Given a set of constraints* $(\mathcal{S}, \mathcal{B})$, *such that* $Simplify(\mathcal{S}, \mathcal{B}) = (\mathcal{S}, \mathcal{B})$, $explicit(\mathcal{S}, \mathcal{B})$ *generates a model* $(\mathcal{E}_0, \mathcal{F}_0, \mathcal{G}_0)$, *for* $(\mathcal{S}, \mathcal{B})$.

Proof: For every boolean constraint $G_{l,i} \in \mathcal{B}$, we assign $\mathcal{G}_0[l, i] = t$. Similarly, for every $F_z \in \mathcal{B}$, we assign $\mathcal{F}_0(z) = t$. The functions $\mathcal{F}_0, \mathcal{G}_0$ map every other element in their domain to f.

Explicit set constraints are of the form: $z \supseteq \lambda x. M$, $z \supseteq C[l, (l_1, \ldots, l_n)]$. They form a regular tree grammar [36]. Since we want to relate the solution of the constraint set to the minimum safe set-based environment, we introduce the constraint $z \supseteq \{-\}$, for every variable z. This is because by the empty rules any subterm can evaluate to $\{-\}$.

There is a least solution to such a collection of explicit constraints. This is computed by constructing an equivalent context-free grammar G with the rules:

- $Z \longrightarrow \lambda x. M$ for the constraint $z \supseteq \lambda x\ M$.

- $Z \longrightarrow C[l, (L_1, \ldots, L_n)]$ for the constraint
  $z \supseteq C[l, (l_1, \ldots, l_n)]$.

- $Z \longrightarrow -$ for every bound variable, every label z.

69

The environment $\mathcal{E}_0$ can then be constructed by mapping every variable z to the set of terms which have finite derivations from the non-terminal Z.

To prove that $(\mathcal{E}_0, \mathcal{F}_0, \mathcal{G}_0)$ is indeed a model we need to show that non-explicit constraints in $(\mathcal{S}, \mathcal{B})$ are not violated. Explicit constraints are, by definition, satisfied by this model. Once we prove that non-explicit set constraints of the form $z_1 \subseteq z_2$ are not violated, it is easy to show that the remaining non-explicit set constraints are not violated. If the constraint $z_1 \subseteq z_2$ is violated, i.e. $z_1 \not\subseteq z_2$, then $\exists v$ st $v \in z_1$ but $v \notin z_2$. But $v \in z_1$ only because we have an explicit set-constraint $z_1 \supseteq ae$, where $v \in ae$. By the rules of the simplification, we also have an explicit constraint $z_2 \supseteq ae$. Hence, violation of a non-explicit constraint of the form $z_1 \subseteq z_2$ is not possible. Using the fact that this explicit constraint is not violated we can easily prove that other non-explicit constraints are not violated.

$\square$

Notation: Given a collection of constraints $(\mathcal{S}, \mathcal{B})$, let $\mathbf{Min}(\mathcal{S}, \mathcal{B})$ denote the model of $(\mathcal{S}, \mathcal{B})$ built in Theorem 6.4.2.

Given a closed term M_0, Table 6.4 presents an algorithm for inferring a collection of constraints, $(\mathcal{S}, \mathcal{B})$, which capture the relation between the elements of the domain of $(\mathcal{E}, \mathcal{F}, \mathcal{G})$, a safe set-based environment wrt M_0.

Theorem 6.4.3 *Given a closed term M_0, there is an $O(n^3)$ algorithm for solving the constraint set $(\mathcal{S}, \mathcal{B})$, such that $M_0 \rhd (\mathcal{S}, \mathcal{B})$, into an explicit form.*

Proof: For a term of size n, i.e. a term with n distinct labels, it is easily seen from the grammar for the constraints and the structure of the constraints generated by *Simplify* that only $O(n^2)$ constraints will be generated during the simplification process.

The simplification process, in essence, performs a dynamic transitive closure: all the 'edges' are not available in the beginning, but generated in the course of the simplification. A non-explicit constraint of the form $l \supseteq \mathrm{Apply}(l_1, l_2)$ cannot be further simplified till we have the explicit boolean constraint F_{l_1} available.

70

$$l : x \;\triangleright\; (\, \{\, l \supseteq x \,\} \,, \{\, F_l \Rightarrow F_x \,\} \,) \tag{1}$$

$$\frac{l_1 : e_1 \;\triangleright\; (\, \mathcal{S}_1 \,, \mathcal{B}_1 \,)}{l : \lambda x\,(l_1 : e_1) \;\triangleright\; (\, \{\, l \supseteq \lambda x\,(l_1 : e_1) \,\} \cup \mathcal{S}_1 \,, \mathcal{B}_1 \,)} \tag{2}$$

$$\frac{l_1 : e_1 \;\triangleright\; (\, \mathcal{S}_1 \,, \mathcal{B}_1 \,) \quad l_2 : e_2 \;\triangleright\; (\, \mathcal{S}_2 \,, \mathcal{B}_2 \,)}{\begin{array}{c} l : \text{letrec } f(x) = (l_1 : e_1) \text{ in } l_2 : e_2 \;\triangleright\; (\; \mathcal{S}_1 \cup \mathcal{S}_2 \cup \{ f \supseteq \{ \lambda x\, l_1 : e_1 \} \} \,, \\ \mathcal{B}_1 \cup \mathcal{B}_2 \cup \{ F_l \Rightarrow F_{l_2} \} \,) \end{array}} \tag{3}$$

$$\frac{l_1 : e_1 \;\triangleright\; (\, \mathcal{S}_1 \,, \mathcal{B}_1 \,) \quad l_2 : e_2 \;\triangleright\; (\, \mathcal{S}_2 \,, \mathcal{B}_2 \,)}{\begin{array}{c} l : (l_1 : e_1)(l_2 : e_2) \;\triangleright\; (\, \{\, l \supseteq \text{Apply}(l_1 \,, l_2) \,\} \cup \mathcal{S}_1 \cup \mathcal{S}_2 \,, \\ \{\, \text{FApply}(l_1 \,, l_2) \,, F_l \Rightarrow F_{l_1} \,\} \cup \mathcal{B}_1 \cup \mathcal{B}_2 \,) \end{array}} \tag{4}$$

$$\frac{l_i : e_i \;\triangleright\; (\, \mathcal{S}_i \,, \mathcal{B}_i \,) \quad i = 1 \dots n}{\begin{array}{c} l : \text{C}(l_1 : e_1 \,, \dots \,, l_n : e_n) \;\triangleright\; (\, \{\, l \supseteq C[l, (l_1 \,, \dots \,, l_n)] \,\} \bigcup_{i=1}^{n} \mathcal{S}_i \,, \\ \{\, F_l \wedge G_{l,i} \Rightarrow F_{l_i} \,\} \bigcup_{i=1}^{n} \mathcal{B}_i \,) \end{array}} \tag{5}$$

$$\frac{l_1 : e_1 \;\triangleright\; (\, \mathcal{S}_1 \,, \mathcal{B}_1 \,) \quad l_2 : e_2 \;\triangleright\; (\, \mathcal{S}_2 \,, \mathcal{B}_2 \,) \quad l_3 : e_3 \;\triangleright\; (\, \mathcal{S}_3 \,, \mathcal{B}_3 \,)}{\begin{array}{c} l : \text{case}(l_1 : e_1 \,, \text{C}(x_1 \,, \dots \,, x_n) \Rightarrow l_2 : e_2 \,, y \Rightarrow l_3 : e_3) \;\triangleright\; \\ (\, \{\, l \supseteq \text{Case}(Y) \,\} \cup \mathcal{S}_1 \cup \mathcal{S}_2 \cup \mathcal{S}_3 \,, \\ \{\, F_l \Rightarrow F_{l_1} \,, \text{FCase1}(Y) \,, \text{FCase2}(Y) \,\} \cup \mathcal{B}_1 \cup \mathcal{B}_2 \cup \mathcal{B}_3 \,) \\ \text{where } Y \equiv (l_1 \,, \text{C}(x_1 \,, \dots \,, x_n) \Rightarrow l_2 \,, y \Rightarrow l_3) \end{array}} \tag{6}$$

Table 6.4 : Construction of Set Constraints

For a graph with n vertices, dynamic transitive closure can be implemented by an $O(n^3)$ algorithm.

□

6.4.2 Relating Set Constraints and Set-Based Semantics

In this subsection we are going to show that $sba_{deadcode}$ is computable. We prove that given a closed term $M_0 \equiv (l_0 : e_0)$, st $M_0 \triangleright (S_0 \,, B_0)$, the set $\{\, l \mid \mathcal{F}_0(l) = false \,, (\mathcal{E}_0, \mathcal{F}_0, \mathcal{G}_0) \equiv Min(S_0 \,, B_0 \cup \{ F_{l_0} \}) \}$ equals $sba_{deadcode}(M_0)$.

This will be proved using the lemmas, Lemma 6.4.1 and Lemma 6.4.2. Lemm 6.4.1 proves that any model of the set of constraints $(S_0 \,, B_0)$, where $M_0 \triangleright (S_0 \,, B_0)$, is a safe

set-based environment for M_0. Lemma 6.4.2 proves the converse, i.e. given a set-based environment $(\mathcal{E}, \mathcal{F}, \mathcal{G})$ safe wrt M_0 we can construct a model for the constraints (S_0, B_0), where $M_0 \triangleright (S_0, B_0)$.

Lemma 6.4.1 *Given a closed term* $M_0 \equiv (l_0 : e_0)$, *st* $M_0 \triangleright (S_0, B_0)$, *if* $(\mathcal{E}, \mathcal{F}, \mathcal{G})$ *is a model of the set of constraints* $(S_0, B_0 \cup \{F_{l_0}\})$, *then* $(\mathcal{E}|_{Var}, \mathcal{F}|_{Var}, \mathcal{G})$ *is safe wrt* M_0, *and* $\{\, l \mid \mathcal{F}(l) = t \,\} \supseteq \mathcal{L}(\mathcal{E}|_{Var}, \mathcal{F}|_{Var}, \mathcal{G})$.

Proof: There are two things to prove here. Given the complete collection of proofs,

$$\{\, \mathcal{E}, \mathcal{F}, \mathcal{G}, t \vdash l_0 : e_0 \rightsquigarrow \dots \,\}$$

Firstly, every such proof is safe and secondly, an occurrence of $\mathcal{E}, \mathcal{F}, \mathcal{G}, t \vdash l : \dots$, implies that $\mathcal{F}(l) = t$.

The proof is by induction on the height of the proof tree.

Induction Hypothesis: For any valid subproof, $\mathcal{E}, \mathcal{F}, \mathcal{G}, t \vdash l : e \rightsquigarrow V$, if $\mathcal{F}(l) = t$ then $V \subseteq \mathcal{E}(l)$ and the subproof satisfies the safety conditions. Also, for every occurrence of, $\mathcal{E}, \mathcal{F}, \mathcal{G}, t \vdash l' : \dots$, we have $\mathcal{F}(l') = t$.

A single case is elaborated below:

Rule 7: Let $M_1 \equiv (l_1 : e_1)$, $M_2 \equiv (l_2 : e_2)$, $M \equiv (l : e)$.

Since the constraint $F_l \Rightarrow F_{l_1}$ is valid, we have $\mathcal{F}(l_1) = t$. Hence, by induction, for the proof of $M_1 \rightsquigarrow V_1$, we have $V_1 \subseteq \mathcal{E}(l_1)$ and the fact that the other conditions in the hypothesis are satisfied. Thus, $\lambda x \, (l_0 : e_0) \in \mathcal{E}(l_1)$. By the definition of FApply, $\mathcal{F}(l_1)$ implies $\mathcal{F}(l_0)$. Hence, $\mathcal{F}(l_0) = t$. By induction, for the proof, $M \rightsquigarrow V_3$, we have that $V_3 \subseteq \mathcal{E}(l_0)$ and the fact that the rest of the conditions in the hypothesis are satisfied. Hence, by the definition of Apply, $V_3 \subseteq \mathcal{E}(l)$.

By the definition of FApply, $\mathcal{F}(l_1) \wedge \mathcal{F}(x)$ implies $\mathcal{F}(l_2)$. If $\mathcal{F}(x)$ is false then the proof of $M_2 \rightsquigarrow \{-\}$ is trivial. If $\mathcal{F}(x)$ is true then $\mathcal{F}(l_2) = t$. Hence, by induction, the proof of $M_2 \rightsquigarrow V_2$ is safe and $V_2 \subseteq \mathcal{E}(l_2)$. Since $(\mathcal{E}, \mathcal{F}, \mathcal{G})$ is a model, we have $\mathcal{E}(x) \supseteq \mathcal{E}(l_2)$. Hence, $\mathcal{E}(x) \supseteq V_2$. Thus making the subproof a safe proof.

$\square$

Lemma 6.4.2 *Given a closed term $M_0 \equiv (l_0 : e_0)$, st $M_0 \triangleright (S_0 , B_0)$, if $(\mathcal{E}, \mathcal{F}, \mathcal{G})$ is safe wrt M_0, then there exists a model $(\mathcal{E}', \mathcal{F}', \mathcal{G}')$ for $(S , B \cup \{F_{l_0}\})$, st $\mathcal{L}(\mathcal{E}, \mathcal{F}, \mathcal{G}) = \{ l \mid \mathcal{F}'(l) = t \}$.*

Proof: Let $\mathcal{E}'|_{Var} = \mathcal{E}$, $\mathcal{F}'|_{Var} = \mathcal{F}$ and $\mathcal{G}' = \mathcal{G}$.

To construct $\mathcal{F}'|_{Label}$, we map every label in $\mathcal{L}(\mathcal{E}, \mathcal{F}, \mathcal{G})$ to t, and any other label to f. The construction of $\mathcal{E}'|_{Label}$ is specified as follows:

I. For any label l, $\mathcal{E}'(l) \supseteq \{-\}$.

II. For any subterm, $(l : x)$, $\mathcal{E}'(l) \supseteq \mathcal{E}(x)$.

III. For any subterm $(l : \lambda x\, M)$, $\mathcal{E}'(l) \supseteq \{\lambda x\, M\}$.

IV. For any subterm $l : C(l_1 : e_1 , \ldots , l_n : e_n)$,
$\mathcal{E}'(l) = \{C[l , (v_1 , \ldots , v_n)] \mid v_i \in \mathcal{E}'(l_i)\}$.

V. For any label l, $\mathcal{E}'(l)$ contains the union of all values V, such that there is a subproof
$\mathcal{E} , \mathcal{F} , \mathcal{G} , t \vdash l : \ldots \rightsquigarrow V$.

The last clause specifies all the relevant bindings. The other clauses are required to satisfy unguarded constraints generated for a closed term. Given $\mathcal{E}$, there is a unique minimum $\mathcal{E}'$ satisfying all clauses except (IV). Once such an $\mathcal{E}'$ is available, we can perform a minimum fixed point computation to make it satisfy clause (IV): this does not violate other clauses.

To complete the proof we need to show that such a construction builds a model satisfying the set constraints. This is easily shown using the safety property of the underlying set-based environment $(\mathcal{E}, \mathcal{F}, \mathcal{G})$.

$\square$

Theorem 6.4.4 (Correctness) *Given a closed term $M_0 \equiv (l_0 : e_0)$, st $M_0 \triangleright (S_0 , B_0)$, if $(\mathcal{E}_0, \mathcal{F}_0, \mathcal{G}_0) \equiv Min(S_0 , B_0 \cup \{F_{l_0}\})$ then the set $\{ l \mid \mathcal{F}_0(l) = f \}$ equals $sba_{deadcode}(M_0)$.*

Proof: Let $(\mathcal{E}'_m, \mathcal{F}'_m, \mathcal{G}'_m)$ be the model constructed from the minimum safe set-based environment, $(\mathcal{E}_m, \mathcal{F}_m, \mathcal{G}_m)$, by an application of Lemma 6.4.2.

By construction, $\mathcal{L}(\mathcal{E}_m, \mathcal{F}_m, \mathcal{G}_m) = \{\, l \mid \mathcal{F}'_m(l) = t \,\}$.

From the construction of the model $(\mathcal{E}_0, \mathcal{F}_0, \mathcal{G}_0)$, as detailed in Theorem 6.4.2, we have $\{\, l \mid \mathcal{F}'_m(l) = t \,\} \supseteq \{\, l \mid \mathcal{F}_0(l) = t \,\}$.

Hence, by Lemma 6.4.2, $\mathcal{L}(\mathcal{E}_m, \mathcal{F}_m, \mathcal{G}_m) \supseteq \{\, l \mid \mathcal{F}_0(l) = t \,\}$ \hfill (1)

By Lemma 6.4.1, $\{\, l \mid \mathcal{F}_0(l) = t \,\} \supseteq \mathcal{L}(\mathcal{E}_0|_{Var}, \mathcal{F}_0|_{Var}, \mathcal{G})$.

As $(\mathcal{E}_m, \mathcal{F}_m, \mathcal{G}_m)$ is the minimum safe set-based environment and the function $\mathcal{L}$ is monotonic, we have

$\mathcal{L}(\mathcal{E}_0|_{Var}, \mathcal{F}_0|_{Var}, \mathcal{G}) \supseteq \mathcal{L}(\mathcal{E}_m, \mathcal{F}_m, \mathcal{G}_m)$.

Hence, $\{\, l \mid \mathcal{F}_0(l) = t \,\} \supseteq \mathcal{L}(\mathcal{E}_m, \mathcal{F}_m, \mathcal{G}_m)$ \hfill (2)

Thus, from (1) and (2), we have,

$\{\, l \mid \mathcal{F}_0(l) = t \,\} = \mathcal{L}(\mathcal{E}_m, \mathcal{F}_m, \mathcal{G}_m)$.

$\square$

6.5   Conclusion

This chapter presents a polynomial time algorithm for isolating dead code in higher-order functional programs, i.e. a terminating algorithm for the computation of static slices. We have presented a formal proof of the correctness of this algorithm.

Analysis concerning demand is essentially a backwards analysis. This work demonstrates that we can still provide an elegant declarative specification of the analysis through an augmented set-based semantics.

The static analysis technique developed here cannot plug space leaks stemming from memo/hash tables [34] or order of evaluation [9]. But, as shown by the example in the Introduction, the analysis can prevent creation of certain void cells and can reduce drag. Further work needs to be done to investigate exactly how significant this optimisation can be.

The language Standard ML contains imperative features, exceptions and assignments. As is to be shown in Chapter 7, introduction of imperative features results in loss of two important properties wrt computation of executable slices:

- Existence of a unique minimum dynamic slice.

- Conceptual understanding of the slice computation process as a 'naive' backward propagation of demand.

Because of the loss of these two properties an immediate extension of the technique, developed in this chapter, to a higher-order functional language, with exceptions and assignments, does not seem feasible.

Chapter 7

Slicing Higher-Order Programs with Exceptions and Assignments

7.1 Slicing Programs with Assignments

The additional productions in the grammar for terms M, is given by

$$
\begin{aligned}
M \quad ::= \quad & l : \,! \, M \\
| \quad & l : \, \mathrm{ref} \, M \\
| \quad & l : \, M_1 := M_2 \\
| \quad & l : \, M_1 \,; \, M_2
\end{aligned}
$$

The statement $l : M_1 \,; \, M_2$ is an abbreviation for $(\lambda z \; M_2) M_1$, where z is not a free variable in M_2. The set of values V, now also contains memory locations β.

As mentioned before, executable slices are interpreted on an augmented operational semantics. Introduction of assignments into the language introduces new operators into the language. We need to augment the standard operational semantics with additional rules for the new operators.

$$
\frac{E \,, \, S \vdash M \; \to \; -, \, S_1}{E \,, \, S \vdash l : \,! \, M \; \to \; -, \, S_1}
$$

$$E\,,\,S\,\vdash\,M_1\,\rightarrow\,-\,,\,S_1$$

$$E\,,\,S_1\,\vdash\,M_2\,\rightarrow\,v\,,\,S_2$$

$$\overline{E\,,\,S\,\vdash\,l\,:\,M_1\,:=\,M_2\,\rightarrow\,(\,)\,,\,S_2}$$

Introduction of assignments complicates the problem of computing the dynamic slice of a program. This is because assignments influence the computation through side-effects: an assignment returns the value ( ) which is not used any further. As discussed in Chapter 1, relevant slices and dynamic slices do not co-incide anymore. With an informal intuition about 'contributing' to the answer, the following items illustrate changes brought about in the computation of dynamic slices by introducing assignments into the language.

The specification of dynamic slices involved changing the environment, as a map from Var $\rightarrow$ Val, to a map from Var $\rightarrow$ Val $* \mathcal{P}$(Labels). If the domain of memory locations is given by the set Loc, we would naturally expect changing the store, as a map from Loc $\rightarrow$ Val, in the standard semantics, to a map from Loc $\rightarrow$ Val $* \mathcal{P}$(Labels). But this alone does not do the trick. For an assignment statement, embedded in a subterm, it may be the case that this containing subterm, by itself, makes no contribution to the value returned, but the contained assignment statement does. The strategy used in Table 5.2 computes the set of labels, defining the dynamic slice, simply from the set of labels collected in its subcomputations. In the presence of assignments, we need to ensure that a particular assignment/update takes place in the sliced version of the program. This cannot be ensured merely by including the label of the particular assignment statement in the dynamic slice. It is necessary to preserve the entire sequence of control dependencies needed to reach the point in the evaluation where the assignment takes place. In the program in Fig 7.1, the entire body of the term `proj1` must be included in the dynamic slice. But a naive extension of the technique described, for purely functional programs, does not ensure that the label attached to `proj1` enters into the memory location marked by y. Hence, at every point in the computation, we need to preserve the set of labels to be left intact, to reach that point in the computation. This set needs to be carried around as an explicit parameter in the natural semantics.

Dynamic slicing, as defined in [6], does not generate an executable program while the

```
let    val x = ref 6
       val y = ref 9
       val proj1 = fn x1 => fn x2 => x1()
in
       (fn() => ! y) ( proj1 (fn() => y:= 90) (fn() => x:=60) )
end
```

Figure 7.1

slicing algorithm defined in [40] does. The difference between the techniques occurs only because statements, in a while loop, are executed many times. As discussed in Section 2.2, Korel and Laski [40], collapse the dependencies of multiple passes of the same statement, through the use of the relation IR, while Agrawal and Horgan [6], do not. A similar problem arises when the same function is used in different call-sites.

In Fig 7.2, the program (a) has (b) as its executable dynamic slice. Fig 7.2(c) is the same program as (a), but with `f1` and `f2` replaced by two different calls to the closure of the same function `f`. In (c), the `if` statement contributes to the value returned by the call (`f x z`) but not to the value returned by the call (`f z x`). The term `p:= false` does not contribute to the value returned by either of the calls to `f`. For purely functional programs, the dynamic slice associated with a closure was the union of the slices associated with each of the contributing call-sites of the closure. Hence, in the dynamic slice of (c), `f` should be defined as,

$$\text{val f = fn y1 => fn y2 =>} \quad (\text{if (! p) then (y1:=100) else } -;$$

$$! \, y1 \,)$$

But this ends up updating the value of `z` in the call (`f z x`). Hence, it no longer returns the same answer. The first call to ! `p` contributes to the final answer but the second call to ! `p` does not. But if enough labels happen to be preserved to reach the second call to ! `p`, in the sliced program, the dependencies, associated with contents of the location `p`, must be also be included in the dynamic slice. Hence, `p := false` must be included in the dynamic slice, even though it does not *explicitly* 'contribute' at any call-site of the containing function.

```
let    val x =    ref 1
       val z =    ref 2
       val p =    ref true              let    val x =    ref −
       val f1 =   fn() => (if (! p)            val z =    ref 2
                  then (x:=100)               val p =    ref true
                  else (z:=200);              val f1 =   fn() => (if (! p)
                  p := false;                            then (x:=100)
                  ! x )                                  else −;
       val f2 =   fn()  => (if (! p)                     ! x )
                  then (z:=100)               val f2 =   fn()  => (! z )
                  else (x:=200);      in
                  p := false;              f1()  +  f2()
                  ! z )               end
  in
      f1()  +  f2()
end

              (a)                                          (b)


let    val x = ref 1
       val z = ref 2
       val p = ref true
       val f = fn y1 => fn y2 =>   ( if (! p) then (y1:=100) else (y2:=200);
                                      p := false;
                                      ! y1 )
  in
      (f x z)  +  (f z x)
end

                                   (c)

                              Figure 7.2




                                79
```

The above example is meant to illustrate a characteristic problem of computing sliced versions of programs, with assignments, which are meant to be executable on the standard/augmented interpreter. The situation being addressed is that of a term, contained in a closure, being executed in two distinct occurrences. One of the occurrences of the term contributes to the final value but the other occurrence does not. In definitions of slicing for term rewriting systems, as formulated in [22], the second occurrence of the term, which does not contribute, is not executed but is side-stepped by the use of the *Resid* relation. But if we are using the standard/augmented interpreter we cannot do this. The problem does not arise in purely functional programs. This is because if we do not preserve, the dependencies associated with the second occurrence of the term, by Lemma 5.2.1, this occurrence will evaluate to $-$. As this occurrence does not contribute, anyway, the rest of the evaluation can proceed as before. But, in the presence of assignments, if the dependencies associated with the second occurrence of the term are not preserved then this occurrence may not evaluate to $-$ but may evaluate to some other value v'. This may happen, for example, if we leave out an update. This different value v' at this point may completely alter the flow of control, resulting in a completely different value being computed.

Another characteristic feature of the introduction of assignments into the language is that *minimum* dynamic slices no longer exist. In Fig 7.3, both the programs, (*b*) and (*c*), are *minimal* dynamic slices of the program (*a*).

It is easy to show that the computation of minimal dynamic slices is no longer decidable. In Fig 7.3(d), the updated value of x depends on the original value of x through some function f. But to know that the update of the variable x is superfluous, we need to execute the rest of the program with old value of x : this execution may not terminate. To exclude the update statement from the dynamic slice we would need to know whether two different inputs to an arbitrary Turing machine, `Turing_Machine1`, produce the same answer: this is an undecidable problem.

```
let    val x = ref 3
in                          let    val x = ref −           let    val x = ref −
       x := 11 ;            in                             in
       x := 16 ;                   x := 16 ;                      x := 11 ;
       (! x) mod 5                 (! x) mod 5                    (! x) mod 5
end                         end                            end

        (a)                         (b)                            (c)
```

```
let    val x = ref 6
       in
       x := f (! x) ;
       Turing_Machine1(!x)
       end
```

(d)

Figure 7.3

A Natural Semantics for Computation of Slices

Table 7.1 presents the specification of dynamic slices for higher-order imperative programs. Proofs in this system are of the form, LL, F, S, $L_0 \vdash M \to V$, S_1, L_1, where

F represents the initial environment with which the computation of M starts.

S represents the initial store with which the computation of the term M starts.

V is the value returned by the computation.

S_1 is the final store.

L_0 is a set of labels representing the control dependencies involved in reaching this particular point in the computation. If any of the labels in L_0 is set to $-$ then this particular point in the computation would never be reached.

LL is a partial function, $LL : \mathcal{P}(\text{Labels}) \to \mathcal{P}(\text{Labels})$.

Let a specific occurrence of a subterm $(l : ! M)$ 'contribute' to the answer. Let there be a *different* occurrence of l, in the execution, such that M evaluates to a location

81

β and the store at that occurrence of l maps β to (V, L). If enough labels are preserved in the slice, so as to reach this occurrence of l, in the execution of the sliced program, then L is included in the dynamic slice, even if this occurrence of l, hence the contents of β, does not contribute to the answer. The rationale for this is: In, at least one occurrence of l, the contents associated with the location, M evaluated to, 'contributed' to the answer. If L, from any one of the other occurrences of l, is not included then it may be that an update associated with the location, M evaluates to at that particular occurrence, will no longer be executed. Consequently, at that occurrence, the accessed value will be different and control may flow in a completely different direction: the sliced program may not terminate or the returned answer of the entire program may be different. This was shown very clearly in the example in Fig 7.2.

Intuitively, LL stores the following information: For an occurrence of the term $(l_0 : ! M)$, if M evaluates to (β, L_1) then $LL(L_1)$ contains the label component, of the contents of the location β. It should be noted that it is easy to synthesize this partial function LL, in the course of the computation. For simpler correctness proofs, LL is included as a constant partial function in the operational semantics. The operational semantics only ensures that, for a given LL, only a subset of evaluations are legal: those that satisfy the side conditions involving LL.

L_1 is a set of labels synthesized during the computation of M. The dynamic slice of the term M is obtained by *closing* the set L_1 with respect to the partial function LL. The rationale behind the closure operation is: if L_1 preserves enough labels to reach a point, in the computation, that involves memory access, then we must include the dependencies, of the contents of the location accessed, even though this specific occurrence does not explicitly 'contribute' to the answer.

We now explain the intuition behind the key rules in the semantics provided in Table 7.1

Rule(4) If $\ldots \Vdash M_1 \rightarrow \langle F', \lambda x \, M \rangle, S_1, L_1$

then the set of labels which need to be preserved to execute the function body M

82

becomes L_1. This becomes relevant when we push label sets into the store during update/ initialisation of memory locations.

Rule(11) If $\ldots \Vdash M_1 \to \beta\,,\,S_1\,,\,L_1$ and $\ldots \Vdash M_2 \to V\,,\,S_2\,,\,L_2$

then the store that is returned has $[\beta \mapsto (V\,,\,L_1 \cup L_2)]$. This is because if the value V, obtained by dereferencing the location β, contributes to the answer then we need to ensure that this update occurs in the sliced program: M_1 must evaluate to β, M_2 must evaluate to V and this point in the computation must be reached.

Rule(12) If $\ldots \Vdash M_1 \to \beta\,,\,S_1\,,\,L_1$ and $S_1(\beta) = (V\,,\,L_2)$

then we need LL to enforce the constraint that if the set of labels L_1 is preserved in the slice, i.e. the term M_1 is executed in the sliced version of the program and evaluates to the location β, then the dereferenced value is the same V.

As discussed in Chapter 4, recursion is implemented by binding the variable representing the recursive function to the recursive closure at each unfolding of the function body. In the semantics presented in Table 7.1, the environment is a mapping from $\mathrm{Var} \to \mathrm{Val} * \mathcal{P}(\mathrm{Labels})$. Hence, at each unfolding of the recursive function, the function variable binding created must have an associated label component. For the Rule 3, the label component is $L_0 \cup \{l\}$. For the Rule 5, the label component is L_1. These are the label sets required to create that particular instance of the recursive closure. An important point to be observed is that at the point of the binding of a recursive closure, the associated label component exactly equals the parameter, on the left of $\Vdash$, representing the set of labels to be retained intact to reach that point in the computation. Because of this, the label set associated with binding of a recursive function variable essentially carries superfluous information. Hence, the label set can be assigned the constant value $\emptyset$. The two rules involving recursive functions may be replaced by the following rules:

$$\frac{LL\,,\,F[f \mapsto (\langle F\,,\,f\,,\,\lambda x\,M_1\rangle\,,\,\emptyset)]\,,\,S_0\,,\,L_0 \cup \{l\}\ \Vdash\ M_2 \to V\,,\,S_1\,,\,L_1}{LL\,,\,F\,,\,S_0\,,\,L_0\ \Vdash\ l:\ \mathrm{letrec}\ f(x) = M_1\ \mathrm{in}\ M_2\ \to\ V\,,\,S_1\,,\,L_1}$$

$$LL,\, F,\, S_0,\, L_0 \cup \{l\} \;\Vdash\; M_1 \;\rightarrow\; \langle F',\, f,\, \lambda x\, M \rangle,\, S_1,\, L_1$$

$$LL,\, F,\, S_1,\, L_0 \cup \{l\} \;\Vdash\; M_2 \;\rightarrow\; V_2,\, S_2,\, L_2$$

$$\dfrac{LL,\, F'[f \mapsto (\langle F',\, f,\, \lambda x\, M \rangle,\, \emptyset),\, x \mapsto (V_2,\, L_2)],\, S_2,\, L_1 \;\Vdash\; M \;\rightarrow\; V_3,\, S_3,\, L_3}{LL,\, F,\, S_0,\, L_0 \;\Vdash\; l : M_1 M_2 \;\rightarrow\; V_3,\, S_3,\, L_3}$$

Proof of Correctness

Definition: For any $L \subseteq \textit{Labels},\ F : \mathcal{P}(\textit{Labels}) \rightarrow \mathcal{P}(\textit{Labels}),\ \ L$ is **closed** under F, i.e. $L = \mathbf{F}^*(L)$, if for any $L_0 \subseteq L,\ F(L_0) \subseteq L$.

The store, at the end of the computation of a sliced term, is very different from the store, at the end of the computation of the original term. This is because the sliced term has many missing assignments and many values have $-$ scattered within them. To prove the correctness of the executable dynamic slice, we need to relate the memory, at the end of the computation of a term, to the memory, at the end of the computation of the sliced version of the term. This is done by the relation $\mathcal{Q}_{\mathcal{L}_1}$.

Definition: For any $\mathcal{L}_1 \subseteq \textit{Labels},\ \mathbf{S}\ \mathcal{Q}_{\mathcal{L}_1}\ \mathbf{S}'$ iff $\ \mathrm{dom}(S') \subseteq \mathrm{dom}(S)\ $ and $\ $ for $l \in \mathrm{dom}(S)$, if $S(l) = (V,\, L)$ then either $(\mathcal{L}_1 \cap L) \neq \emptyset$, or $(\mathcal{L}_1 \cap L) = \emptyset\ $ and $\ S'(l) = V[\mathcal{L}_1/-]$.

Given the domain and the contents of a store S, for any S' such that $S\ \mathcal{Q}_{\mathcal{L}_1}\ S'$, the relationship $\mathcal{Q}_{\mathcal{L}_1}$ specifies a minimum domain and associated contents for S'.

Lemma 7.1.1 *Let* $\ LL,\, F,\, S,\, L_0 \;\Vdash\; (l : e) \;\rightarrow\; V,\, S_1,\, L_1$.
If there is a location β s.t. $S(\beta)$ is not defined or $S(\beta) \not\equiv S_1(\beta)$, and $\ S_1(\beta) = (V,\, L)$ then $\ L_0 \cup \{l\} \subseteq L$.

Proof: The proof is a simple induction on the height of the proof tree, using the fact that if $\ LL,\, E,\, S,\, L_0 \;\Vdash\; (l : e) \;\rightarrow\; V,\, S_1,\, L_1\ $ then $\ L_0 \cup \{l\} \subseteq L_1$.

$\square$

$$LL\,,\,F[x \mapsto (V\,,\,L)]\,,\,S_0\,,\,L_0 \;\Vdash\; l:\; x \;\rightarrow\; V\,,\,S\,,\,L_0 \cup L \cup \{l\} \tag{1}$$

$$LL\,,\,F\,,\,S_0\,,\,L_0 \;\Vdash\; l:\; \lambda x\,M \;\rightarrow\; \langle F\,,\,\lambda x\,M\rangle\,,\,S\,,\,L_0 \cup \{l\} \tag{2}$$

$$\frac{LL\,,\,F[f \mapsto (\langle F\,,\,f\,,\,\lambda x\,M_1\rangle\,,\,L_0 \cup \{l\})]\,,\,S_0\,,\,L_0 \cup \{l\} \;\Vdash\; M_2 \rightarrow V\,,\,S_1\,,\,L_1}{LL\,,\,F\,,\,S_0\,,\,L_0 \;\Vdash\; l:\; \mathrm{letrec}\; f(x) = M_1 \;\mathrm{in}\; M_2 \;\rightarrow\; V\,,\,S_1\,,\,L_1} \tag{3}$$

$$\frac{\begin{array}{c}LL\,,\,F\,,\,S_0\,,\,L_0 \cup \{l\} \;\Vdash\; M_1 \;\rightarrow\; \langle F'\,,\,\lambda x\,M\rangle\,,\,S_1\,,\,L_1 \\ LL\,,\,F\,,\,S_1\,,\,L_0 \cup \{l\} \;\Vdash\; M_2 \;\rightarrow\; V_2\,,\,S_2\,,\,L_2 \\ LL\,,\,F'[x \mapsto (V_2\,,\,L_2)]\,,\,S_2\,,\,L_1 \;\Vdash\; M \;\rightarrow\; V_3\,,\,S_3\,,\,L_3\end{array}}{LL\,,\,F\,,\,S_0\,,\,L_0 \;\Vdash\; l:\; M_1 M_2 \;\rightarrow\; V_3\,,\,S_3\,,\,L_3} \tag{4}$$

$$\frac{\begin{array}{c}LL\,,\,F\,,\,S_0\,,\,L_0 \cup \{l\} \;\Vdash\; M_1 \;\rightarrow\; \langle F'\,,\,f\,,\,\lambda x\,M\rangle\,,\,S_1\,,\,L_1 \\ LL\,,\,F\,,\,S_1\,,\,L_0 \cup \{l\} \;\Vdash\; M_2 \;\rightarrow\; V_2\,,\,S_2\,,\,L_2 \\ LL\,,\,F'[f \mapsto (\langle F'\,,\,f\,,\,\lambda x\,M\rangle\,,\,L_1)\,,\,x \mapsto (V_2\,,\,L_2)]\,,\,S_2\,,\,L_1 \;\Vdash\; M \;\rightarrow\; V_3\,,\,S_3\,,\,L_3\end{array}}{LL\,,\,F\,,\,S_0\,,\,L_0 \;\Vdash\; l:\; M_1 M_2 \;\rightarrow\; V_3\,,\,S_3\,,\,L_3} \tag{5}$$

$$\frac{LL\,,\,F\,,\,S_{i-1}\,,\,L_0 \cup \{l\} \;\Vdash\; M_i \;\rightarrow\; V_i\,,\,S_i\,,\,L_i\,,\; i = 1 \ldots n}{LL\,,\,F\,,\,S_0\,,\,L_0 \;\Vdash\; l:\; \mathrm{Op}(M_1\,,\,\ldots\,,\,M_n) \;\rightarrow\; \mathrm{Op}(V_1\,,\,\ldots\,,\,V_n)\,,\,S_n\,,\,\bigcup_{i=1}^{n} L_i} \tag{6}$$

$$\frac{LL\,,\,F\,,\,S_{i-1}\,,\,L_0 \cup \{l\} \;\Vdash\; M_i \;\rightarrow\; V_i\,,\,S_i\,,\,L_i\,,\; i = 1 \ldots n}{LL\,,\,F\,,\,S_0\,,\,L_0 \;\Vdash\; l:\; \mathrm{C}(M_1\,,\,\ldots\,,\,M_n) \;\rightarrow\; C((V_1\,,\,L_1)\,,\,\ldots\,,\,(V_n\,,\,L_n))\,,\,S_n\,,\,L_0 \cup \{l\}} \tag{7}$$

$$\frac{\begin{array}{c}LL\,,\,F\,,\,S_0\,,\,L_0 \cup \{l\} \;\Vdash\; M_1 \;\rightarrow\; C((V_1\,,\,L_1)\,,\,\ldots\,,\,(V_n\,,\,L_n))\,,\,S_1\,,\,L \\ LL\,,\,F[x_1 \mapsto (V_1\,,\,L_1)\,,\,\ldots\,,\,x_n \mapsto (V_n\,,\,L_n)]\,,\,S_1\,,\,L \;\Vdash\; M_2 \;\rightarrow\; V\,,\,S_2\,,\,L'\end{array}}{LL\,,\,F\,,\,S_0\,,\,L_0 \;\Vdash\; l:\; \mathrm{case}(M_1\,,\,\mathrm{C}(x_1\,,\,\ldots\,,\,x_n) \Rightarrow M_2\,,\,y \Rightarrow M_3) \;\rightarrow\; V\,,\,S_2\,,\,L'} \tag{8}$$

$$\frac{\begin{array}{c}LL\,,\,F\,,\,S_0\,,\,L_0 \cup \{l\} \;\Vdash\; M_1 \;\rightarrow\; C'((V_1\,,\,L_1)\,,\,\ldots\,,\,(V_n\,,\,L_n))\,,\,S\,,\,L \quad C \neq C' \\ LL\,,\,F[y \mapsto C'((V_1\,,\,L_1)\,,\,\ldots\,,\,(V_n\,,\,L_n))]\,,\,S\,,\,L \;\Vdash\; M_3 \;\rightarrow\; V\,,\,S'\,,\,L'\end{array}}{LL\,,\,F\,,\,S_0\,,\,L_0 \;\Vdash\; l:\; \mathrm{case}(M_1\,,\,\mathrm{C}(x_1\,,\,\ldots\,,\,x_n) \Rightarrow M_2\,,\,y \Rightarrow M_3) \;\rightarrow\; V\,,\,S'\,,\,L'} \tag{9}$$

$$\frac{LL\,,\,F\,,\,S_0\,,\,L_0 \cup \{l\} \;\Vdash\; M \;\rightarrow\; V\,,\,S_1\,,\,L_1 \quad \beta \notin \mathrm{dom}(S_1)}{LL\,,\,F\,,\,S_0\,,\,L_0 \;\Vdash\; l:\; \mathrm{ref}\,M \;\rightarrow\; \beta\,,\,S_1[\beta \mapsto (V\,,\,L_1)]\,,\,L_0 \cup \{l\}} \tag{10}$$

$$\frac{\begin{array}{c}LL\,,\,F\,,\,S_0\,,\,L_0 \cup \{l\} \;\Vdash\; M_1 \;\rightarrow\; \beta\,,\,S_1\,,\,L_1 \\ LL\,,\,F\,,\,S_1\,,\,L_0 \cup \{l\} \;\Vdash\; M_2 \;\rightarrow\; V\,,\,S_2\,,\,L_2\end{array}}{LL\,,\,F\,,\,S_0\,,\,L_0 \;\Vdash\; l:\; M_1 := M_2 \;\rightarrow\; ()\,,\,S_2[\beta \mapsto (V\,,\,L_1 \cup L_2)]\,,\,L_0 \cup \{l\}} \tag{11}$$

$$\frac{LL\,,\,F\,,\,S_0\,,\,L_0 \cup \{l\} \;\Vdash\; M_1 \;\rightarrow\; \beta\,,\,S_1\,,\,L_1}{LL\,,\,F\,,\,S_0\,,\,L_0 \;\Vdash\; l:\,!\,M_1 \;\rightarrow\; V\,,\,S_1\,,\,L_1 \cup L_2} \tag{12}$$
$$\text{where } S_1(\beta) = (V\,,\,L_2) \text{ and } L_2 \subseteq LL(L_1)$$

Table 7.1 : Specifying Dynamic Slices for Higher-Order Imperative Programs

Lemma 7.1.2 *If* LL, F, S_0, $L_0 \Vdash (l : e) \rightarrow V$, S_1, L_1 *then for any* $\mathcal{L}$, *s.t.*
$L_0 \subseteq \mathcal{L}$ *and* $\mathcal{L}$ *is closed under LL, if* $\mathcal{L}_1 \equiv (Labels - \mathcal{L})$ *and* $S \, \mathcal{Q}_{\mathcal{L}_1} \, S'$ *then*
$F[\mathcal{L}_1/-]$, $S' \vdash (l : e)[\mathcal{L}_1/-] \rightarrow (V, L_1)[\mathcal{L}_1/-]$, S_1', *where* $S_1 \, \mathcal{Q}_{\mathcal{L}_1} \, S_1'$.

Proof: The proof is by induction on the height of the proof tree. Only the proof for the memory access rule, Rule (12), uses the closure property of $\mathcal{L}$. All the cases considered in the proof assume that $l \in \mathcal{L}$. If this wasn't true then $l \in \mathcal{L}_1$ and
$F[\mathcal{L}_1/-]$, $S' \vdash l : - \rightarrow -$, S'. By Lemma 7.1.1, any update/allocation made in the evaluation of $(l : e)$, contains l in its label component. Hence, $S_1 \, \mathcal{Q}_{\mathcal{L}_1} \, S'$.

Rule(4) As $l \notin \mathcal{L}_1$ then $\mathcal{L}_1 \cap (L_0 \cup \{l\}) = \emptyset$. If $\mathcal{L}_1 \cap L_1 = \emptyset$ then, by induction,

$\quad F[\mathcal{L}_1/-]$, $S' \vdash M_1[\mathcal{L}_1/-] \rightarrow \langle F'[\mathcal{L}_1/-], \lambda x \, M[\mathcal{L}_1/-] \rangle$, S_1', where $S_1 \, \mathcal{Q}_{\mathcal{L}_1} \, S_1'$.

$\quad$ By induction, $F[\mathcal{L}_1/-]$, $S_1' \vdash M_2[\mathcal{L}_1/-] \rightarrow (V_2, L_2)[\mathcal{L}_1/-]$, S_2', where $S_2 \, \mathcal{Q}_{\mathcal{L}_1} \, S_2'$.

$\quad$ By assumption $\mathcal{L} \cap L_1 = \emptyset$. Hence, by induction,

$\quad F'[\mathcal{L}_1/-][x \mapsto (V_2, L_2)[\mathcal{L}_1/-]]$, $S_2' \vdash M[\mathcal{L}_1/-] \rightarrow (V_3, L_3)[\mathcal{L}_1/-]$, S_3',

$\quad$ where $S_3 \, \mathcal{Q}_{\mathcal{L}_1} \, S_3'$.

$\quad$ Hence, $F[\mathcal{L}_1/-]$, $S' \vdash (l : M_1 M_2)[\mathcal{L}_1/-] \rightarrow (V_3, L_3)[\mathcal{L}_1/-]$, S_3'.

If $\mathcal{L}_1 \cap L_1 \neq \emptyset$ then, by induction, $\quad F[\mathcal{L}_1/-]$, $S' \vdash M_1[\mathcal{L}_1/-] \rightarrow -$, S_1',
where $S_1 \, \mathcal{Q}_{\mathcal{L}_1} \, S_1'$.

$\quad$ By induction, $F[\mathcal{L}_1/-]$, $S_1' \vdash M_2[\mathcal{L}_1/-] \rightarrow (V_2, L_2)[\mathcal{L}_1/-]$, S_2', where $S_2 \, \mathcal{Q}_{\mathcal{L}_1} \, S_2'$.

$\quad$ Hence, $F[\mathcal{L}_1/-]$, $S' \vdash (l : M_1 M_2)[\mathcal{L}_1/-] \rightarrow -$, S_2'. Any updates/allocations made in the evaluation of M necessarily introduces L_1 into the store. Hence, any location, in which S_2 and S_3 differ, contains at least L_1 in the label component. Since $\mathcal{L}_1 \cap L_1 \neq \emptyset$ and we are comparing stores modulo $\mathcal{Q}_{\mathcal{L}_1}$, $S_3 \, \mathcal{Q}_{\mathcal{L}_1} \, S_2'$.

Rule(8) As $l \notin \mathcal{L}_1$ then $\mathcal{L}_1 \cap (L_0 \cup \{l\}) = \emptyset$. If $\mathcal{L} \cap L_1 = \emptyset$ then, by induction,

$\quad F[\mathcal{L}_1/-]$, $S' \vdash M_1[\mathcal{L}_1/-] \rightarrow C((V_1, L_1)[\mathcal{L}/-], \dots, (V_n, L_n)[\mathcal{L}/-])$, S_1', where
$S_1 \, \mathcal{Q}_{\mathcal{L}_1} \, S_1'$. By assumption, $\mathcal{L} \cap L_1 = \emptyset$. Hence, by induction,

$$F[\mathcal{L}_1/-][x_1 \mapsto (V_1, L_1)[\mathcal{L}/-], \dots, x_n \mapsto (V_n, L_n)[\mathcal{L}/-]], S_1' \vdash$$

$$M_2[\mathcal{L}_1/-] \rightarrow (V, L')[\mathcal{L}_1/-], S_2'$$

where $S_2 \, \mathcal{Q}_{\mathcal{L}_1} \, S_2'$.

If $\mathcal{L}_1 \cap L_1 \neq \emptyset$ then, by induction, $\quad F[\mathcal{L}_1/-]$, $S' \vdash M_1[\mathcal{L}_1/-] \rightarrow -$, S'_1, where $S_1 \mathcal{Q}_{\mathcal{L}_1} S'_1$. Any updates/allocations made in the evaluation of M_2 necessarily introduces L_1 into the store. Hence, any location, in which S_2 and S_1 differ, contains at least L_1 in the label component. Since $\mathcal{L}_1 \cap L_1 \neq \emptyset$ and we are comparing stores modulo $\mathcal{Q}_{\mathcal{L}_1}$, $S_2 \mathcal{Q}_{\mathcal{L}_1} S'_1$.

Rule(10) As $l \notin \mathcal{L}_1$, by induction, $F[\mathcal{L}_1/-]$, $S' \vdash M[\mathcal{L}_1/-] \rightarrow (V, L_1)[\mathcal{L}_1/-]$, S'_1, where $S_1 \mathcal{Q}_{\mathcal{L}_1} S'_1$. Hence, $S_1[\beta \mapsto (v, L_1)] \mathcal{Q}_{\mathcal{L}_1} S'_1[\beta \mapsto (V, L_1)[\mathcal{L}_1/-]]$. This uses the assumption that $\text{dom}(S'_1) \subseteq \text{dom}(S_1)$, i.e. if β is a completely new location in the proof of the evaluation of $(\text{ref } M)$ under S_0 then β is also a completely new location in the proof of the evaluation of $(\text{ref } M)[\mathcal{L}_1/-]$ under S'.

Rule(11) As $l \notin \mathcal{L}_1$, i.e. $\mathcal{L}_1 \cap (L_0 \cup \{l\}) = \emptyset$ then, by induction,

$F[\mathcal{L}_1/-]$, $S' \vdash M_1[\mathcal{L}_1/-] \rightarrow (\beta, L_1)[\mathcal{L}_1/-]$, S'_1, where $S_1 \mathcal{Q}_{\mathcal{L}_1} S'_1$.

$F[\mathcal{L}_1/-]$, $S'_1 \vdash M_2[\mathcal{L}_1/-] \rightarrow (V_2, L_2)[\mathcal{L}_1/-]$, S'_2, where $S_2 \mathcal{Q}_{\mathcal{L}_1} S'_2$.

If $(L_1 \cup L_2) \cap \mathcal{L}_1 = \emptyset$ then $M_1[\mathcal{L}_1/-]$ evaluates to β and $M_2[\mathcal{L}_1/-]$ evaluates $V[\mathcal{L}_1/-]$. Thus $S_2[\beta \mapsto (V, L_1 \cup L_2)] \mathcal{Q}_{\mathcal{L}_1} S'_2[\beta \mapsto V[\mathcal{L}_1/-]]$.

If $(L_1 \cup L_2) \cap \mathcal{L}_1 \neq \emptyset$ then the updated version of S_2, $S_2[\beta \mapsto (V, L_1 \cup L_2)]$, is definitely related by $\mathcal{Q}_{\mathcal{L}_1}$ to a version of S'_2, which possibly differs from S_2' only in the location β.

Rule(12) If $\mathcal{L}_1 \cap L_1 = \emptyset$, i.e. $L_1 \subseteq \mathcal{L}$ then, by induction,

$F[\mathcal{L}_1/-]$, $S' \vdash M_1 \rightarrow \beta$, S'_1, where $S_1 \mathcal{Q}_{\mathcal{L}_1} S'_1$.

Since $L_1 \subseteq \mathcal{L}$ and $\mathcal{L}$ is closed under LL, $L_2 \subseteq LL(L_1) \subseteq \mathcal{L}$. Hence, $L_2 \cap \mathcal{L}_1 = \emptyset$.

As $S_1 \mathcal{Q}_{\mathcal{L}_1} S'_1$, $\beta \in \text{dom}(S_1')$ and $S'_1(\beta) = V[\mathcal{L}_1/-]$. Hence,

$E[\mathcal{L}_1/-]$, $S' \vdash l :! M_1 \rightarrow V[\mathcal{L}_1/-]$, S'_1, where $S_1 \mathcal{Q}_{\mathcal{L}_1} S'_1$.

If $l \notin \mathcal{L}_1$ and $\mathcal{L}_1 \cap L_1 \neq \emptyset$ then, by induction,

$F[\mathcal{L}_1/-]$, $S' \vdash M_1 \rightarrow -$, S'_1, where $S_1 \mathcal{Q}_{\mathcal{L}_1} S'_1$.

Hence, $F[\mathcal{L}_1/-]$, $S' \vdash l :! M_1 \rightarrow -$, S'_1.

It is important to notice that the closure of the set $\mathcal{L}$ under the function LL ensures that if the set L_1 is retained, i.e. M_1 evaluates to the same location, then the set of labels, L_2, associated with its contents is retained.

87

□

Theorem 7.1.1 *If LL, $\emptyset$, $\emptyset$, $\emptyset \Vdash M \rightarrow V$, S, L then $LL^*(L)$ is a dynamic slice.*

Proof: $LL^*(L)$ is a set of labels that is closed under the function LL.

Let $\mathcal{L}_1 \equiv \Lambda(M) - LL^*(L)$. Since $\mathcal{L}_1 \cap L = \emptyset$, by Lemma 7.1.2,

$\emptyset$, $\emptyset \vdash M[\mathcal{L}_1/-] \rightarrow V[\mathcal{L}_1/-]$, S', where $S \; \mathcal{Q}_{\mathcal{L}_1} \; S'$.

Hence, $LL^*(L)$ defines an executable dynamic slice of the program M.

□

7.2   Slicing Programs with Exceptions

In this section, we investigate the slicing of purely functional programs in the presence of exceptions.

The additional productions in the grammar for labelled terms M, is given by

M ::= $l : e$

e ::=   let exception D in M

| M_1 handle $l_2 : (\mathrm{D}(x_1, \ldots, x_n) \Rightarrow M_2)$

| raise M

Since we denote deletion of a subterm by substitution with $-$, the deletion of an exception handler leaves behind a term of the form "$l_1 : M_1$ handle $l_2 : -$". For a simpler presentation we will assume, when required, that such terms are post-processed to $l_1 : M_1$.

In SML, exceptions are generative[47] by nature. The set of values V, now also contains exception values of the form $[\delta, ((V_1, L_1), \ldots, (V_n, L_n))]$. Exception values are similar to constructor values discussed earlier. Unlike ordinary constructors, exception constructors are assigned a unique value every time the declaration is evaluated.

The environment F, in addition to mapping variables to the set of values V, also maps exception constructors to elements from the set Δ. The substitution function $[\mathcal{L}/-]$ is identity on exception constructor mappings: $[\mathrm{D} \mapsto \delta][\mathcal{L}/-] = [\mathrm{D} \mapsto \delta]$.

In the presence of exceptions, in the labelled calculus, terms can evaluate to (V, L) or to an exception packet of the form $\ll [\delta, ((V_1, L_1), \ldots, (V_n, L_n))], L \gg$.

Because of the fact that exceptions radically alter the flow of control in a program, it is easy to show that minimum dynamic slices no longer exist and even computation of minimal dynamic slices is undecidable. The following program is written in an SML-like syntax:

let exception A of int

in (fn x => 5) (raise (A 5)) handle (A x) => x end

The program can be sliced to:

(fn x => 5)—

Or,

let exception A of int

in —(raise (A 5)) handle (A x) => x end

Just as in the presence of assignments, collecting labels, which directly 'contribute' to the answer, does not generate a dynamic slice. We need to close this collected label set with respect to a synthesized function.

let   exception A of int
         fun F f f′ = f (f′ 3) ((f′ 4) handle (A x) => x)
         fun g x y = x + y
         fun g′ x = x + 3
         fun h x y = x + 40
         fun h′ x = if (x=3) then 45 else raise(A 60)
in
         (F g g′) + (F h h′) + (h′ 4 handle (A x) => x + 75)
end

Figure 7.4

In the call (F g g′), in the program in Fig 7.4, all subterms in the body of F, except the handle expression, directly 'contribute' to the returned answer. The call (h′ 4 handle (A x) => x + 75) is added to ensure that the expression (raise(A 60)), in h′, directly contributes to the answer. In the call (F h h′), since h is a function that only uses its first argument, a strategy that collects labels that only directly 'contribute'

89

to the answer will fail to collect the label of the `handle` expression in `F`. But if the `handle` expression in `F` is not included in the dynamic slice then the call (`F h h'`) will end up raising an uncaught exception.

```
let    exception A of int
       exception B of int
       fun F f f' = f (f' 3) (#2((f' 4),(f' 5)) handle (A x) => x)
       fun g x y = x + y
       fun g' x = if (x=5) then raise(A (x+2)) else x + 3
       fun h x y = x + 40
       fun h' x = if (x=3) then 5 else if (x=4) then raise(A 6) else raise(B 7)
in
       (F g g') + (F h h')  + (h' 4 + h' 5 handle (A x) => x + 9)
end
```

Figure 7.5

For the program in Fig 7.5, since (`f' 4`) is the first argument to a second projection function, a strategy which only collects labels which directly 'contribute', will not collect its label. But if (`f' 4`) is not included, in the dynamic slice, then in the call, (`F h h'`), (`h' 5`) will be evaluated and will raise an uncaught exception `B`. In the evaluation of the unsliced program, (`h' 5`) was not evaluated because (`h' 4`) raised the exception `A`.

The wisdom gained, from these examples, is the following: whenever any label l, which evaluates to an exception packet, or any dynamic dependencies of l, are deleted then we must necessarily delete the labelled term, which contains the handler, that transforms the exception packet, that l evaluates to, into a value. This is because if the exception is never raised, in the sliced program, then segments of the program may be executed that were never executed before. Similarly, the deletion of any exception handler, which catches an exception packet, may result in the modified program raising an uncaught exception or executing a handler which was not executed before. Hence, deletion of a handler, which actually catches an exception packet, must involve the deletion of the term in which it is immediately contained. This is essentially a closure condition. As in the case with assignments, this closure condition needs to be introduced because of a function closure

having multiple contributing call-sites.

A Natural Semantics for the Computation of Slices

The additional set of rules for computing the dynamic slice of a functional program, with exceptions, is given in Table 7.2. The rest of the rules are very similar to those in Table 7.1, except that the partial function LL is replaced by the partial function KK and the store S is replaced by the parameter Ex. Additionally, there are rules introduced by the exception convention discussed in Chapter 4.

KK is a partial function from $\mathcal{P}(\text{Labels})$ to $\mathcal{P}(\text{Labels})$, storing information concerning the closure condition discussed in the introduction. For the term $l : M_1 \text{ handle } l_2 : (\text{D}(x_1, \ldots, x_n) \Rightarrow M_2)$, let M_1 evaluate to an exception packet, which is caught by the handler labelled l_2. Let L_0 be the set of labels of terms to be retained intact to reach the point in the computation where the evaluation of the term l begins. $KK(L_0 \cup \{l\})$ contains the label l_2 as well the dynamic dependencies of the exception packet caught. This closure condition forms a part of Rule 5.

Note: $KK, F, Ex, L_0 \Vdash M \rightarrow \ldots$ is considered well-formed if the value of any exception constructor in F is contained in Ex. Henceforth, we will only be talking about well-formed terms.

Lemma 7.2.1 *For any set of labels $\mathcal{L}$, such that $\mathcal{L}$ is closed under KK and $L_0 \subseteq \mathcal{L}$, let $\mathcal{L}_1 \equiv (\text{Labels} - \mathcal{L})$.*

If $Ex' \subseteq Ex$ and $KK, F, Ex, L_0 \Vdash l : e \rightarrow V, Ex_1, L_1$ then

$F[\mathcal{L}_1/-], Ex' \vdash (l : e)[\mathcal{L}_1/-] \rightarrow (V, L_1)[\mathcal{L}_1/-], Ex_1{}', \text{ where } Ex_1{}' \subseteq Ex_1.$

If $Ex' \subseteq Ex, \mathcal{L}_1 \cap L' = \emptyset$ and $KK, F, Ex, L_0 \Vdash l : e \rightarrow \ll [\delta, \overline{(V_i, L_i)}], L' \gg, Ex_1$ then

$F[\mathcal{L}_1/-], Ex' \vdash (l : e)[\mathcal{L}_1/-] \rightarrow \ll [\delta, \overline{(V_i, L_i)}[\mathcal{L}_1/-]] \gg, Ex_1{}', \text{ where } Ex_1{}' \subseteq Ex_1.$

Proof: The proof is by induction on the height of the proof tree. The closure property of KK is only used for Rule(5): the rule that translates an exception packet to a value.

$$\frac{KK\,,\,F\,,\,Ex_0\,,\,L_0 \cup \{l\} \;\Vdash\; M \;\to\; [\delta\,,\,\overline{(V_i\,,\,L_i)}]\,,\,Ex_1\,,\,L}{KK\,,\,F\,,\,Ex_0\,,\,L_0 \;\Vdash\; l:\,\mathrm{raise}\,M \;\to\; \ll [\delta\,,\,\overline{(V_i\,,\,L_i)}]\,,\,L \gg\,,\,Ex_1} \tag{1}$$

$$\frac{KK\,,\,F[D \mapsto \delta]\,,\,Ex_{i-1}\,,\,L_0 \cup \{l\} \;\Vdash\; M_i \;\to\; V_i\,,\,Ex_i\,,\,L_i \quad i = 1 \ldots n}{KK\,,\,F[D \mapsto \delta]\,,\,Ex_0\,,\,L_0 \;\Vdash\; l:\,\mathrm{D}(M_1\,,\,\ldots M_n) \;\to\; [\delta\,,\,\overline{(V_i\,,\,L_i)}]\,,\,Ex_n\,,\,L_0 \cup \{l\}} \tag{2}$$

$$\frac{KK\,,\,F\,,\,Ex_0\,,\,L_0 \cup \{l\} \;\Vdash\; M_1 \;\to\; V\,,\,Ex_1\,,\,L_1}{KK\,,\,F\,,\,Ex_0\,,\,L_0 \;\Vdash\; l:\,M_1\,\mathrm{handle}\,l_2:\,(\mathrm{D}(x_1\,,\,\ldots\,,\,x_n) \Rightarrow M_2) \;\to\; V\,,\,Ex_1\,,\,L_1} \tag{3}$$

$$\frac{KK\,,\,F\,,\,Ex_0\,,\,L_0 \cup \{l\} \;\Vdash\; M_1 \;\to\; \ll [\delta\,,\,\overline{(V_i\,,\,L_i)}]\,,\,L' \gg\,,\,Ex_1}{\begin{array}{c}KK\,,\,F\,,\,Ex_0\,,\,L_0 \;\Vdash\; l:\,M_1\,\mathrm{handle}\,l_2:\,(\mathrm{D}(x_1\,,\,\ldots\,,\,x_n) \Rightarrow M_2) \\ \to\; \ll [\delta\,,\,\overline{(V_i\,,\,L_i)}]\,,\,L' \gg\,,\,Ex_1 \\ \mathrm{where}\ F(\mathrm{D}) \;\equiv\; \delta' \neq \delta\end{array}} \tag{4}$$

$$\frac{\begin{array}{c}KK\,,\,F\,,\,Ex_0\,,\,L_0 \cup \{l\} \;\Vdash\; M_1 \;\to\; \ll [\delta\,,\,\overline{(V_i\,,\,L_i)}]\,,\,L' \gg\,,\,Ex_1 \\ KK\,,\,F[x_1 \mapsto (V_1\,,\,L_1)\,,\,\ldots\,,\,x_n \mapsto (V_n\,,\,L_n)]\,,\,Ex_1\,,\,L' \cup \{l_2\} \;\Vdash\; M_2 \;\to\; V\,,\,Ex_2\,,\,L\end{array}}{\begin{array}{c}KK\,,\,F\,,\,Ex_0\,,\,L_0 \;\Vdash\; l:\,M_1\,\mathrm{handle}\,l_2:\,(\mathrm{D}(x_1\,,\,\ldots\,,\,x_n) \Rightarrow M_2) \;\to\; V\,,\,Ex_2\,,\,L \\ \mathrm{where}\quad L' \cup \{l_2\} \subseteq KK(L_0 \cup \{l\}) \\ F(\mathrm{D}) = \delta\end{array}} \tag{5}$$

$$\frac{KK\,,\,F[D \mapsto \delta]\,,\,Ex_0 \cup \{\delta\}\,,\,L_0 \cup \{l\} \;\Vdash\; M \;\to\; V_1\,,\,Ex_1\,,\,L_1 \quad \mathrm{where}\ \delta \notin Ex}{KK\,,\,F\,,\,Ex_0\,,\,L_0 \;\Vdash\; l:\,\mathrm{let\ exception\ D\ in}\,M \;\to\; V_1\,,\,Ex_1\,,\,L_1} \tag{6}$$

Table 7.2 : Specifying Dynamic Slices in the Presence of Exceptions

Rule(5) There are two cases to consider here:

- If $\mathcal{L}_1 \cap L = \emptyset$ then $\mathcal{L}_1 \cap L' = \emptyset$. This is because $L' \subseteq L$. Hence, by induction,

  $F[\mathcal{L}_1/-]$, $Ex' \vdash M[\mathcal{L}_1/-] \to \ll [\delta, \overline{(V_i, L_i)}[\mathcal{L}_1/-]] \gg, Ex_1'$,

  where $Ex_1' \subseteq Ex_1$.

  Since $l_2 \in L$, the handler has not been deleted. By induction, the execution of the handler returns $V[\mathcal{L}_1/-]$, Ex_2' , where $Ex_2' \subseteq Ex_2$.

- $\mathcal{L}_1 \cap L \neq \emptyset$. If $l \in \mathcal{L}_1$ then, obviously, l evaluates to $(-, Ex_0')$.

  If $l \notin \mathcal{L}_1$ then $l \in \mathcal{L}$. As $L_0 \subseteq \mathcal{L}$ and $\mathcal{L}$ is closed under KK, $L' \cup \{l_2\} \subseteq KK(L_0 \cup \{l\}) \subseteq \mathcal{L}$, i.e $(L' \cup \{l_2\}) \cap \mathcal{L}_1 = \emptyset$. Hence, by induction,

  $F[\mathcal{L}_1/-]$, $Ex' \vdash M[\mathcal{L}_1/-] \to \ll [\delta, \overline{(V_i, L_i)}[\mathcal{L}_1/-]] \gg, Ex_1'$.

  As the handler l_2 has not been deleted, we have, by induction,

  $$F[x_1 \mapsto (V_1, L_1)[\mathcal{L}_1/-], \ldots, x_n \mapsto (V_n, L_n)[\mathcal{L}_1/-]], Ex_1' \vdash$$

  $$M_2[\mathcal{L}_1/-] \to -, Ex_2'$$

  These are cases which assume that the evaluation of the handler results in a value and not an exception packet. On the contrary, if the evaluation of the handler returns $\ll [\delta_2, \{(v_2, L_2)\}], L'' \gg, Ex_2'$ then $L' \cup \{l_2\} \subseteq L''$. Hence, if $\mathcal{L}_1 \cap L'' = \emptyset$ then $\mathcal{L}_1 \cap (L' \cup \{l_2\}) = \emptyset$. We can now apply the induction hypothesis to get the requisite result.

Let us now look at the function application rule, in the instance in which one of the antecedents evaluates to an exception packet.

$$KK, F, Ex_0, L_0 \cup \{l\} \Vdash M_1 \to \langle F', \lambda x\, M \rangle, Ex_1, L_1$$

$$KK, F, Ex_1, L_0 \cup \{l\} \Vdash M_2 \to V_2, Ex_2, L_2$$

$$\frac{KK, F'[x \mapsto (V_2, L_2)], Ex_2, L_1 \Vdash M \to V_3, Ex_3, L_3}{KK, F, Ex_0, L_0 \Vdash l : M_1 M_2 \to V_3, Ex_3, L_3}$$

If all the antecedents in this rule evaluate to a value instead of an exception packet then the proof of correctness is the same as that in the purely functional case.

But the term $(l : M_1 M_2)$ may evaluate to an exception packet, $\ll [\delta, (v, L)], L' \gg$. This may happen in one of three possible ways:

- If KK, F, Ex_0, $L_0 \cup \{l\}$ $\Vdash$ M_1 $\to \ll [\delta, \overline{(V_i, L_i)}], L' \gg$, Ex then

  KK, F, Ex_0, L_0 $\Vdash$ $l : M_1 M_2$ $\to \ll [\delta, \overline{(V_i, L_i)}], L' \gg$, Ex.

  By induction, $F[\mathcal{L}_1/-]$, Ex_0' $\vdash$ $M_1[\mathcal{L}_1/-]$ $\to \ll [\delta, \overline{(V_i, L_i)}[\mathcal{L}_1/-]] \gg$, Ex'.

- If M_1 evaluates to a value but,

  KK, F, Ex_1, $L_0 \cup \{l\}$ $\Vdash$ M_2 $\to \ll [\delta, \overline{(V_i, L_i)}], L' \gg$, Ex_2,

  by induction, $F[\mathcal{L}_1/-]$, Ex_1' $\Vdash$ $M_2[\mathcal{L}_1/-]$ $\to \ll [\delta, \overline{(V_i, L_i)}[\mathcal{L}_1/-]] \gg$, Ex_2'.

- Both the function and the argument evaluate to values but the application may generate an exception packet. Since, $\mathcal{L}_1 \cap L_3 = \emptyset$ and $L_1 \subseteq L_3$, $\mathcal{L}_1 \cap L_1 = \emptyset$. Hence, by induction, $F[\mathcal{L}_1/-]$, Ex_0 $\vdash$ $M_1[\mathcal{L}_1/-]$ $\to \langle F'[\mathcal{L}_1/-], \lambda x\, M[\mathcal{L}_1/-] \rangle$, Ex_1'.

  By induction, $F[\mathcal{L}_1/-]$, Ex_1' $\vdash$ $M_2[\mathcal{L}_1/-]$ $\to (V_2, L_2)[\mathcal{L}_1/-]$, Ex_2'.

  Since, KK, $F'[x \mapsto (V_2, L_2)]$, Ex_2, L_1 $\Vdash$ M $\to \ll [\delta, (V, L)], L_3 \gg$, Ex_3,

  by induction,

  $$F'[\mathcal{L}_1/-][x \mapsto (V_2, L_2)[\mathcal{L}_1/-]], Ex_2', L_1 \vdash M[\mathcal{L}_1/-] \to \ll [\delta, (V, L)[\mathcal{L}_1/-]], Ex_3' \gg$$

  $\square$

Theorem 7.2.1 *If* KK, $\emptyset$, $\emptyset$, $\emptyset$ $\Vdash$ M $\to V$, Ex, L

then $KK^*(L)$ *is a dynamic slice of* M.

If KK, $\emptyset$, $\emptyset$, $\emptyset$ $\Vdash$ M $\to \ll [\delta, \overline{(V_i, L_i)}], L \gg$, Ex

then $KK^*(L)$ *is a dynamic slice of* M.

Proof: $KK^*(L)$ is a set of labels that is closed under the function KK.

Let $\mathcal{L}_1 \equiv \Lambda(l : M) - KK^*(L)$. Since $\mathcal{L}_1 \cap L = \emptyset$, by Lemma 7.2.1,

$\emptyset$, $\emptyset$ $\vdash l : M[\mathcal{L}_1/-]$ $\to V[\mathcal{L}_1/-]$, Ex'.

Hence, $KK^*(L)$ defines a dynamic slice of the program M.

Similarly, if M evaluates to an exception packet, by Lemma 7.2.1,

$\emptyset$, $\emptyset$ $\vdash$ $M[\mathcal{L}_1/-]$ $\to \ll [\delta, \overline{(V_i, L_i)}[\mathcal{L}_1/-]] \gg$, Ex'. $\square$

7.3   Integrating Assignments and Exceptions

It turns out that, from the point of view of slicing, exceptions and assignments are orthogonal issues. Hence, the specification of slicing, in a language with exceptions, can

be merged with the specification of slicing, in a language with assignments. The proof of correctness of this merged specification is essentially a merger of the two correctness proofs. The specification of dynamic slices, in the presence of assignments, uses a global partial function LL. The specification, in the presence of exceptions, uses a global partial function KK. It can be shown that closing the returned set of labels with respect to the functions LL and KK, generates a dynamic slice. We have the following theorems:

Lemma 7.3.1 *If* KK, LL, F, S, Ex, $L_0 \Vdash M \rightarrow V$, S_1, Ex_1, L_1 *then*
for any $\mathcal{L}$, *s.t.* $L_0 \subseteq \mathcal{L}$ *and* $\mathcal{L}$ *is closed under* LL *and under* KK,

If $\mathcal{L}_1 \equiv (Labels - \mathcal{L})$, $S \mathcal{Q}_{\mathcal{L}_1} S'$ *and* $Ex' \subseteq Ex$ *then*
$$F[\mathcal{L}_1/-], S', Ex' \vdash M[\mathcal{L}_1/-] \rightarrow (V, L_1)[\mathcal{L}_1/-], S_1', Ex_1'$$
$$\text{where } S_1 \mathcal{Q}_{\mathcal{L}_1} S_1', Ex_1' \subseteq Ex_1$$

If $\mathcal{L}_1 \cap L' = \emptyset$ *and* KK, LL, F, S, Ex, $L_0 \Vdash M \rightarrow \ll [\delta, \overline{(V_i, L_i)}], L' \gg, S_1, Ex_1$
then $F[\mathcal{L}_1/-], S', Ex' \vdash M[\mathcal{L}_1/-] \rightarrow \ll [\delta, \overline{(V_i, L_i)}[\mathcal{L}_1/-]] \gg, S_1', Ex_1'$
where $S_1 \mathcal{Q}_{\mathcal{L}_1} S_1', Ex_1' \subseteq Ex_1$

The above lemma can be proved by using a lemma built from the combination of Lemma 7.1.2 and Lemma 7.2.1.

Theorem 7.3.1 *If* KK, LL, $\emptyset$, $\emptyset$, $\emptyset$, $\emptyset \Vdash M \rightarrow V$, S, Ex, L
then $(KK \sqcup LL)^*(L)$ *is a dynamic slice of* M.
If KK, LL, $\emptyset$, $\emptyset$, $\emptyset$, $\emptyset \Vdash M \rightarrow \ll [\delta, \overline{(V_i, L_i)}], L \gg, S, Ex$
then $(KK \sqcup LL)^*(L)$ *is a dynamic slice of* M.

7.4 Broader Slicing Criteria

In his seminal paper, Weiser [67], defines slicing with respect to a two parameter slicing criterion: a set of variables and a statement number. In contrast, Reps and Yang[55], use a single parameter slicing criterion: the 'behavior' of a statement in a program. Venkatesh [63], and Agrawal and Horgan[6], also use a single parameter slicing criterion: the value of a variable at the end of the evaluation of a program. Till date, for higher-order programs,

we have been using a fixed slicing criteria: the value computed by a program. We now investigate as to how we can move to a two parameter slicing criteria similar to Weiser's.

The operational definition of a program slice developed by Weiser uses the concept of a projection on a state trajectory of an evaluation. Such a projection for a slicing criterion, $C \equiv \langle i, x \rangle$, can be generated by a special `print` statement placed right before statement i, of the program. A `print` statement in a program can be viewed as dynamic allocation of data in a special memory. Data is never accessed from this memory nor is it ever updated. The store S in the specification of dynamic slices, is partitioned into the old store S and the new special memory, *Stream*. The proof rules remain the same: rules carry $(S, Stream)$ instead of S alone. Rules which modified S, now modify the S component. We have a new rule for `print` statements:

$$\frac{LL, F, S, Stream, L_0 \cup \{l\} \Vdash M_1 \rightarrow V, S_1, Stream_1, L_1}{LL, F, S, L_0 \Vdash l: \text{print } M_1 \rightarrow (), S_1, (V, L_1) :: Stream_1, L_0 \cup \{l\}}$$

We will assume that print statements can legally print out only nullary constructors.

Note

- As *Stream* is a memory that is never accessed, it is constructed as a list of values instead of a partial map from the set of locations to values. When it is more convenient to consider *Stream* as a partial map instead of a list, we will be implicitly doing so.

- It is memory access through the ! operator that generates constraints for the partial function LL. *Stream* is a memory that is never accessed during the execution of a program and hence, does not generate any constraints for LL.

Definition: Given a term M, with $(l_0 : e_0)$ as a subterm and $C \equiv \langle l_0, x \rangle$ as a slicing criterion, let the evaluation of $M' \equiv M[l_0/(l_{00} : \text{print } x); (l_{01} : e_0)]$ return the stream, $Stream_1$. A set of labels $\mathcal{L}$ defines the **dynamic slice with respect to slicing criterion** C, if for $\mathcal{L}_1 \equiv \Lambda(l : e) - \mathcal{L}$, the evaluation of $M'[\mathcal{L}_1/-]$ also returns $Stream_1$.

If the evaluation of a term returns a stream $[(v_1, L_1), \ldots, (v_n, L_n)]$, then by an augmented version of Lemma 7.1.2, we can show that the closure of $(L_1 \cup \ldots \cup L_n)$ with

respect to LL is the dynamic slice with respect to the slicing criterion.

Theorem 7.4.1 *Let* $Stream_1 \equiv [(V_1, L_1), \ldots, (V_n, L_n)]$ *and* $C \equiv \langle l_0, x \rangle$.

If $LL, \emptyset, \emptyset, \emptyset, \emptyset \Vdash M \to V, S_1, Stream_1, L_0$

then $LL^*(L_1 \cup \ldots \cup L_n)$ *defines a slice with respect to the slicing criterion* C.

    Proof: Let $\mathcal{L} = LL^*(L_1 \cup \ldots \cup L_n)$ and $\mathcal{L}_1 = \Lambda(M) - \mathcal{L}$.

If $LL, \emptyset, \emptyset, \emptyset, \emptyset \Vdash M \to V, S_1, Stream_1, Ex_1, L_0$

then, by Lemma 7.1.2, $\vdash M[\mathcal{L}_1/-] \to (V, L_0)[\mathcal{L}_1/-], S_1{}', Stream_1{}',$

where $S_1 \mathcal{Q}_{\mathcal{L}_1} S_1{}'$, $Stream_1 \mathcal{Q}_{\mathcal{L}_1} Stream_1{}'$. As $\mathcal{L}_1 \cap (L_1 \cup \ldots \cup L_n) = \emptyset$,

$\mathrm{dom}(Stream_1) = \mathrm{dom}(Stream_1{}')$. Since V_i is a first-order value, we have

$Stream_1{}' \equiv [V_1, \ldots, V_n]$.

$\square$

    It is easy to extend this proof to a language with exceptions.

Extensional Characterisations

We do not have an algorithm-independent characterisation of the slice we compute. We would like to investigate whether we can provide a denotational characterisation or a collecting interpretation on a lazy semantics, along the lines of Hudak and Young [33].

Chapter 8

Program Instrumentation and an Implementation Overview

The earlier chapters of this thesis presented natural deduction proof systems for the computation of an executable dynamic slice of a higher-order program. A naive but labor-intensive way of implementing the specification is to write a special interpreter, for labelled SML programs, which mimics the computation of the proof system. In this chapter, we show that it is possible to translate a labelled program into an unlabelled program such that when this code executes on the standard interpreter, for unlabelled programs, it returns a tuple: the first component of the tuple is the value returned, by the execution of the labelled program, and the second component is the set of labels, to be closed with respect to global functions, to obtain an executable dynamic slice.

The aim of the empirical analysis is to obtain an estimate of the size of the dynamic slice of a program, with respect to a particular input, in comparison to the the execution slice of the program. Given an Core-SML program, the strategy used is as follows: we first attach unique labels to the node of the parse tree of the program and then translate this labelled program to an unlabelled program. This unlabelled program is executed on the standard interpreter and it returns the expected answer as well as the set of labels representing the dynamic slice.

8.1 Program Instrumentation

In this section we provide a syntax-directed translation for transforming a labelled program into an unlabelled SML program. The translator introduces a host of bound variables in the let-constructs and abstractions. In an actual implementation, these variables are going to be given uniques names, by the use of some kind of a "gensym" function, within the translator. But, for a simple exposition of the translation, a name generator is not used while introducing bound variables: the names of variables are picked from a fixed finite set.

A call to the translation function $[\![\ ]\!]$ is of the form $[\![M]\!]_{L,R}$ where:

- M is a labelled term to be translated.

- L is a variable name whose value is a set of labels. The entire set of control dependencies, required to reach a program point, is explicitly carried along through the computation, in the operational semantics presented in the previous chapters. This is represented by the set of labels L, to the left of $\Vdash$ in the semantics given in Table 7.1 and Table 7.2. The label set L is essentially an attribute that is inherited from previous terms executed in the computation. The computation and propagation of this label set is handled by passing the name of the variable, containing the current label set, as the second parameter to the translator function. An alternate approach would be to use a two parameter translator function which translates every term into an abstraction which takes in a set of labels as input. But this would create too many redexes, at run-time, which could be reduced statically.

- The third parameter R to the translator function is a set of variables. This is required for handling recursive functions. The semantics for the computation of dynamic slices, as discussed in Chapter 7, has environments which are finite maps, $\text{Var} \rightarrow \text{Val} * \mathcal{P}(\text{Labels})$. This means that a variable, representing a recursive function, should be bound to a tuple whose second component is a label set. But bindings for recursive functions are created automatically by the compiler from the `let` construct and cannot be simulated using any other mechanism. Hence, a variable representing a recursive function gets bound to an abstraction and

not to a tuple, consisting of an abstraction and a set of labels, as other function variables are. Fortunately, as mentioned in Section 7.1, the label set associated with a recursive closure may be assigned the constant value $\emptyset$. The strategy used by the translator function is to pass around the set of variables, which are bound by recursive function definitions and are currently *in-scope*, as the third parameter. To translate $[\![l : x]\!]_{L,R}$, we first check whether x is a variable bound in an in-scope recursive function definition by checking membership in R. If it is, then we know that x is an abstraction, not a tuple, and we return $(x, L \cup \{l\})$. Otherwise x must be bound to a tuple (V, L_1) and we return $(V, L_1 \cup L \cup \{l\})$.

I. $[\![l : x]\!]_{L,R} \;=\; \text{if } (x \in R)$

$\qquad\qquad\qquad\quad then \;\; (x, L \cup \{l\})$

$\qquad\qquad\qquad\quad else \;\; \text{let } (v, L_1) = x \;\text{ in }\; (v, L \cup L_1 \cup \{l\})\,)\;\; \text{end}$

II. $[\![l : \lambda x\, M]\!]_{L,R} \;=\; \text{let } f = \lambda x\, \lambda L_1\, [\![M]\!]_{L_1,R}$

$\qquad\qquad\qquad\quad \text{in } \; (f, L \cup \{l\}) \;\; \text{end}$

III. $[\![l : \text{letrec } f(x) = M_1 \text{ in } M_2]\!]_{L,R} \;=\; \text{let } f(x) = \lambda L_1\, [\![M_1]\!]_{L_1, R \cup \{f\}}$

$\qquad\qquad\qquad\qquad\qquad\qquad L_2 = L \cup \{l\}$

$\qquad\qquad\qquad\qquad\qquad \text{in } [\![M_2]\!]_{L_2, R \cup \{f\}} \;\; \text{end}$

IV. $[\![l : M_1\, M_2]\!]_{L,R} \;=\; \text{let } L_0 = L \cup \{l\}$

$\qquad\qquad\qquad\qquad\qquad (f, L_1) = [\![M_1]\!]_{L_0, R}$

$\qquad\qquad\qquad\qquad\qquad V = [\![M_2]\!]_{L_0, R}$

$\qquad\qquad\qquad\qquad \text{in } \; f\, V\, L_1 \;\; \text{end}$

V. $[\![l : \text{Op}(M_1, \ldots, M_n)]\!]_{L,R} \;=\; \text{let } L_0 = L \cup \{l\}$

$\qquad\qquad\qquad\qquad\qquad\quad \{\, (V_i, L_i) = [\![M_i]\!]_{L_0, R} \,\}$

$\qquad\qquad\qquad\qquad\qquad \text{in } \; (\, Op(V_1, \ldots V_n), \cup_{i=1}^{n} L_i\,) \;\; \text{end}$

VI. $[\![l : \text{C}(M_1, \ldots, M_n)]\!]_{L,R} \;=\; \text{let } L_1 = L \cup \{l\}$

$\qquad\qquad\qquad\qquad\qquad \text{in } \; (\, C([\![M_1]\!]_{L_1, R}, \ldots, [\![M_n]\!]_{L_1, R}), L_1\,) \;\; \text{end}$

100

VII. $[\![\, l :\ \mathrm{case}(M_1,\ \mathrm{C}(x_1,\ \ldots,\ x_n) \Rightarrow M_2,\ y \Rightarrow M_3)\,]\!]_{L,R}\ =$

$\qquad$ let $L_0 = L \cup \{l\}$

$\qquad$ in $\quad$ case $(\,[\![\, M_1\,]\!]_{L_0,R}\,)$ of $\quad (\,\mathrm{C}(x_1,\ \ldots,\ x_n),\ L_1\,) => [\![\, M_2\,]\!]_{L_1,R}$

$\qquad\qquad\qquad\qquad\qquad\qquad |\ (\,y,\ L_1\,) => [\![\, M_3\,]\!]_{L_1,R}$

$\qquad$ end

VIII. $[\![\, l :\ \mathrm{ref}\, M\,]\!]_{L,R}\ =\quad$ let $L_0 = L \cup \{l\}$

$\qquad\qquad\qquad\qquad\qquad$ in $(\,\mathrm{ref}\,(\,[\![\, M\,]\!]_{L_0,R}\,),\ L_0\,)$ end

IX. $[\![\, l :!\, M\,]\!]_{L,R}\ =\quad$ let $L_0 = L \cup \{l\}$

$\qquad\qquad\qquad\qquad\qquad (ll,\ L_1) = [\![\, M\,]\!]_{L_0,R}$

$\qquad\qquad\qquad\qquad\qquad (V,\ L_2) = !\, ll$

$\qquad\qquad\qquad\qquad\qquad \_ = \mathtt{Update}(MM,\ L_1,\ L_2)$

$\qquad\qquad\qquad\qquad$ in $(V,\ L_1 \cup L_2)$ end

X. $[\![\, l :\ M_1 := M_2\,]\!]_{L,R}\ =\quad$ let $L_0 = L \cup \{l\}$

$\qquad\qquad\qquad\qquad\qquad\qquad (ll,\ L_1) = [\![\, M_1\,]\!]_{L_0,R}$

$\qquad\qquad\qquad\qquad\qquad\qquad (V,\ L_2) = [\![\, M_2\,]\!]_{L_0,R}$

$\qquad\qquad\qquad\qquad\qquad$ in $(\,ll := (V,\ L_1 \cup L_2),\ L_0\,)$ end

XI. $[\![\, l :\ \text{let exception D in } M\,]\!]_{L,R}\ =\quad$ let exception D

$\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad L_0 = L \cup \{l\}$

$\qquad\qquad\qquad\qquad\qquad\qquad\qquad$ in $[\![\, M\,]\!]_{L_0,R}$ end

XII. $[\![\, l :\ \mathrm{raise}\, M\,]\!]_{L,R}\ =\quad$ let $L_0 = L \cup \{l\}$

$\qquad\qquad\qquad\qquad\qquad\qquad$ in raise SPECIAL_E$(\,[\![\, M\,]\!]_{L_0,R}\,)$ end

XIII. $[\![\, l \, : \, M_1 \text{ handle } l_2 \, : \, (\mathrm{D}(x_1 \, , \, \ldots x_n) \Rightarrow M_2) \,]\!]_{L \, , \, R} \; =$

$\quad\quad$ let $L_0 = L \cup \{l\}$

$\quad\quad$ in $[\![\, M_1 \,]\!]_{L_0 \, , \, R}$ handle $\quad \text{SPECIAL\_E}(\mathrm{D}(x_1 \, , \, \ldots x_n) \, , \, L_1) \Rightarrow$

$\quad\quad\quad\quad\quad$ let $L_2 = L_1 \cup \{l_2\}$

$\quad\quad\quad\quad\quad\quad$ $y = [\![\, M_2 \,]\!]_{L_2 \, , \, R}$

$\quad\quad\quad\quad\quad\quad\quad$ $\_ = \text{Update}(MM \, , \, L_0 \, , \, L_2)$

$\quad\quad\quad\quad\quad$ in y end

$\quad\quad$ end

As discussed in Chapter 7, the semantics for the computation of dynamic slices, in the presence of imperative features, uses global constant partial functions, LL and KK. The translated version of the labelled program synthesizes these functions in the course of the computation. Instead of having subcomputations returning their own synthesized fragments of LL and KK and taking their join, we introduce a single global variable MM which is updated in the course of the computation. At the end of the computation, the set of labels returned need to be closed with respect to this global function. The construction of the partial function MM uses a function Update with the following semantics:

$$\text{Update}(MM \, , \, M \, , \, M') = \begin{cases} MM := MM \sqcup (M \mapsto M') & \text{if } M \notin \text{dom}(MM) \\ MM := MM \sqcup (M \mapsto MM(M) \cup M') & \text{otherwise} \end{cases}$$

In the instrumented code for $(l \, : \, \text{raise} \, (l_1 \, : \, M_1))$, $(l_1 \, : \, M_1)$ evaluates to a tuple consisting of a value of exception type and a set of labels. But the term labelled l needs to raise an exception. Hence, all raise expressions raise the same exception SPECIAL_E applied to the instrumented argument. By pattern matching on the argument, to the exception constructor SPECIAL_E, handlers can find out the actual raised exception. If there is a match with the exception constructor, in the argument to SPECIAL_E, the handler begins to execute. If the set of labels is given the ML type Set_of_Labels then the exception constructor SPECIAL_E is declared as follows,

```
exception SPECIAL_E of exn * Set_of_Labels;
```

where `exn` is the type for exceptions in SML.

In Rule XIII: the translation for terms containing exception handlers, it is important to note that an update to the global function MM is logged only if M_2, the body of the exception handler, evaluates to a value and not an exception packet.

An issue concerning exceptions has been side-stepped in the translation process. Atomic operations on first-order constants, represented by Op, can raise exceptions in SML/NJ. Division by zero raises the exception `Div`. Addition and Multiplication can raise the exception `Overflow`. Operations on real numbers can raise the exception `BadReal`. All these exceptions must be caught and translated to the exception `SPECIAL_E of exn * Set_of_Labels`. Hence, for every atomic operation, we need to define a handler that catches all possible exceptions, that can be raised by the operation, and re-raises these as the exception `SPECIAL_E` along with the label set of dependencies.

8.1.1 Correctness of Program Instrumentation

To compute the dynamic slice of a labelled term M, it is first translated to an unlabelled term M_1 using the translator function.

Let $\quad M_1 \quad \equiv \quad$ let $L = \emptyset$

$$\quad\quad\quad\quad\quad \text{in} \;\; [\![\, M \,]\!]_{L,\emptyset} \;\; \text{end}$$

The following sequence of statements are then executed on the SML interpreter:

$> val\; MM = \text{ref}\; \emptyset\;;$

$> exception\; SPECIAL\_E\; of\; exn * Set\_of\_Labels\;;$

$> val\; (v\,,\,L) = M_1\;;$

After this computation, the set L is closed with respect to the function MM to obtain the dynamic slice.

The correctness of the instrumentation is stated by Theorem 8.1.1. For simplicity, we assume that there are no recursive functions. Recursive functions do not add to the technical complexity of the proof: we only need a stronger hypothesis to ensure that any closure, associated with a recursive function, is included in the the set R, which is used in the instrumentation function $[\![\;\;]\!]_{L,R}$.

103

We need to define a relation $\mathcal{A}$, relating values in the semantics for dynamic slicing, as specified in Table 7.1, with values in the standard semantics, as generated by the annotated version of the program.

- $(\langle F ,\, \lambda x\, M \rangle ,\, L_0)\ \mathcal{A}\ (\langle E ,\, \lambda x \lambda L\, [\![\, M\,]\!]_{L ,\, \emptyset} \rangle ,\, L_0)$ if $F\ \mathcal{A}\ E$.

- $(C(V_1 ,\, \ldots ,\, V_n) ,\, L_0)\ \mathcal{A}\ (C(U_1 ,\, \ldots ,\, U_n) ,\, L_0)$ if $\forall i,\, V_i\ \mathcal{A}\ U_i$.

- $F[x \mapsto (V ,\, L)]\ \mathcal{A}\ E[x \mapsto (U ,\, L)]$ if $(F\ \mathcal{A}\ E)$ and $(V ,\, L)\ \mathcal{A}\ (U ,\, L)$.

- $[\,]\ \mathcal{A}\ E$: This relation is present to accommodate for the fact that the annotated program has extra bound variables for Label sets.

- The relation $\mathcal{A}$, for environments, is similarly extended to relate stores.

The natural semantics for the computation of dynamic slices, as presented in Table 7.1, uses a constant relation LL for capturing control dependencies arising out of assignments and exceptions. The annotated program constructs this relation, during evaluation, using a reference variable $\mathtt{MM}$. It is easy to show that at any point in the computation, the value of $\mathtt{MM}$ is consistent with LL, i.e. if $LL(L_1) = L_2$ then $!\mathtt{MM}(L_1) \subseteq L_2$. Since, at the end of the computation $\mathtt{MM}$ satisfies every constraint which LL does and is also consistent with it, $!\mathtt{MM}$ can be taken to a possible value of LL.

Theorem 8.1.1 *Given a labelled term* $(l :\, e)$*, if*
$F\ \mathcal{A}\ E$ *,* $S_0\ \mathcal{A}\ R_0$ *and* $LL ,\, F ,\, S_0 ,\, L_0\ \Vdash\ (l :\, e)\ \rightarrow V ,\, S_1 ,\, L_1$ *then*
$E[K_0 \mapsto L_0] ,\, R_0\ \vdash\ [\![\, (l :\, e)\,]\!]_{K_0 ,\, \emptyset}\ \rightarrow (U ,\, L_1) ,\, R_1,$
where $(V ,\, L)\ \mathcal{A}\ (U ,\, L)$ *and* $S_1\ \mathcal{A}\ R_1$*.*

Proof: The proof is constructed by induction on the height of the proof-tree, based on the semantics in Table 7.1. The simplicity of the proof arises from the fact that the slice, constructed in Table 7.1, is essentially the computation of a synthesized attribute of the proof tree representing the standard semantics. The annotated version of the program simply introduces code to compute this synthesized attribute during the evaluation of the program.

Here, we go through one important case:

Rule (App):

$$LL \, , \, F \, , \, S_0 \, , \, L_0 \cup \{l\} \; \Vdash \; M_1 \; \rightarrow \; \langle F' \, , \, \lambda x \, M \rangle \, , \, S_1 \, , \, L_1$$

$$LL \, , \, F \, , \, S_1 \, , \, L_0 \cup \{l\} \; \Vdash \; M_2 \; \rightarrow \; V_2 \, , \, S_2 \, , \, L_2$$

$$\dfrac{LL \, , \, F'[x \mapsto (V_2 \, , \, L_2)] \, , \, S_2 \, , \, L_1 \; \Vdash \; M \; \rightarrow \; V_3 \, , \, S_3 \, , \, L_3}{LL \, , \, F \, , \, S_0 \, , \, L_0 \; \Vdash \; l : \; M_1 M_2 \; \rightarrow \; V_3 \, , \, S_3 \, , \, L_3}$$

The annotation version of the term $M_1 \, M_2$, $[\![\, l : \; M_1 \, M_2 \,]\!]_{K_0 \, , \, \emptyset}$, is

$$\texttt{let } K_1 = K_0 \cup \{l\}$$

$$(f \, , \, K_2) = [\![\, M_1 \,]\!]_{K_1 \, , \, \emptyset}$$

$$V = [\![\, M_2 \,]\!]_{K_1 \, , \, \emptyset}$$

$$\texttt{in } f \, V \, K_2 \texttt{ end}$$

By induction, we have,

$E[K_0 \mapsto L_0 \, , \, K_1 \mapsto L_0 \cup \{l\}] \, , \, R_0 \; \vdash \; [\![\, M_1 \,]\!]_{K_1 \, , \, \emptyset} \; \rightarrow \; (\langle E' \, , \, \lambda x \, \lambda K [\![\, M \,]\!]_{K \, , \, \emptyset} \rangle \, , \, L_1) \, , \, R_1$

where $F' \, \mathcal{A} \, E'$ and $S_1 \, \mathcal{A} \, R_1$.

$E[K_0 \mapsto L_0 \, , \, K_1 \mapsto L_0 \cup \{l\}] \, , \, R_1 \; \vdash \; [\![\, M_2 \,]\!]_{K_1 \, , \, \emptyset} \; \rightarrow \; (U_2 \, , \, L_2) \, , \, R_2$

where $(V_2 \, , \, L_2) \, \mathcal{A} \, (U_2 \, , \, L_2)$ and $S_2 \, \mathcal{A} \, R_2$.

Hence, $E'[x \mapsto (U_2 \, , \, L_2) \, , \, K \mapsto L_1 \; \vdash \; [\![\, M \,]\!]_{K \, , \, \emptyset} \; \rightarrow \; (U_3 \, , \, L_3)] \, , \, R_3$

where $(V_3 \, , \, L_3) \, \mathcal{A} \, (U_3 \, , \, L_3)$ and $S_3 \, \mathcal{A} \, R_3$.

$\square$

8.2   Implementation Details

We have implemented, in SML'93, a program which takes in a core SML program and returns an annotated version of the program. We have also implemented a support program to calculate the execution trace of a program. The implementation is built using the parser, elaborator and other libraries from the ML Kit [13].

8.2.1   The Interface of the Annotating Program

Our implementation provides a single function,

```
type L = Set_of_Labels_Static
final_answer:  string list -> L * L * L
```

The argument to the function is a list of strings, with the following elements:

— Element 1: The name of the file in which the program to be annotated resides.

— Element 2: The name of the file in which the annotated program is to be dumped.

— Element 3: The name of the file in which program execution statistics and slice size information is to be saved.

— Element 4[Optional]:   The name of the file in which a listing of node labels, which form a part of the execution slice but not a part of the dynamic slice, is to be preserved.

The value returned by the function call is a triple of label sets: the first component is the set forming the dynamic slice, the second component is the set forming the execution slice and the third is the set consisting of the entire set of labels in the program.

Given a function call, `final_answer ["p.sml","p.annotated", "p.log","p.diff"]` , we now present details regarding the structure of the input and output files.

The Structure of the Input File

Assumptions regarding the input file `"p.sml"` are detailed below:

- The program contained in `"p.sml"` should be a core SML program. The annotated version of `"p.sml"` is saved in the file `"p.annotated"`. The file `"p.annotated"` contains a valid SML program that type checks. Hence, we need a translation for datatype declarations and explicit types present in the input program:

  $[\![b]\!]$ $\implies$ $b * \mathtt{Set\_of\_Labels}$, where b is a basic type or a datatype/abstype.

  $[\![\tau_1 \rightarrow \tau_2]\!]$ $\implies$ $([\![\tau_1]\!] \rightarrow \mathtt{Set\_of\_Labels} \rightarrow [\![\tau_2]\!]) * \mathtt{Set\_of\_Labels}$

  $[\![\tau_1 * \tau_2]\!]$ $\implies$ $([\![\tau_1]\!] * [\![\tau_2]\!]) * \mathtt{Set\_of\_Labels}$

  After the parsing of the program in `"p.sml"` is complete, every node in the parse tree has an associated label which is a 4-tuple of integers: the first two integers give

the line number and the column number, in the input file, where the node begins and the next two integers give the line number and column number where the node ends. The type `Set_of_Labels_Static` is the abstract type for such a set, whose elements are of the type `int*int*int*int`. Since sets of labels are constructed at run-time, efficiency is of utmost importance. Hence, a 4-tuple representing the label of a node is mapped to an integer. The type `Set_of_Labels` is the abstract type for such a set of integers.

- It is assumed that the program contained in `"p.sml"` does not contain eqtype variables and that the equality operator operator is restricted to terms of basic types. SML provides for a built-in polymorphic equality operator. We cannot generate an annotated version of this operator because there is no source-level implementation of this operator. Nor is putting some kind of a wrapper around the available equality operator a viable strategy. This is because a pair $(1, 2)$, in the annotated program, evaluates to a value $(((1, L_1), (2, L_2)), L_0)$. As per the semantics of the original program, this pair, in the annotated program, should be considered to be equal to any other annotated pair value of the form $(((1, L_1'), (2, L_2')), L_0')$.

- For a program returning a value v, of a basic type `'b`, the annotated program returns a value (v, L) where the set L, after closing it with respect to relevant relations, is the executable dynamic slice for the computation of the answer. But, for a program returning a value, of a non-basic type, the annotated program returns a value of much more complicated internal structure.

Let the program in the input file be returning a list `[1,2]`. The datatype `list`, in the annotated program, is given by the following declaration:

```
datatype 'a list = nil
  | :: of (('a*Set_of_Labels)*('a list*Set_of_Labels))*Set_of_Labels
```

The program-user may desire to perform a slice computation to isolate the set of labels which were 'needed' in the computation of every element of the list or the set of labels which were 'needed' in the computation of a specific element of the list. To provide the user the ability to specify a slicing criterion, in a most general way, we require the input program to terminate with a function

107

```
val w =  8 + 9

val x = 10

val z = (fn y => y) w

val ANSWER = fn f => f (makestring  z)
```

Figure 8.1: The sample program `p.sml`

```
ANSWER: (string -> unit) -> unit.

                    val ANSWER = fn f =>  ...
```

We would like the user to treat the argument function `f` just like the standard output function (`IO.outputc std_out`), in SML/NJ, and apply this function `f` to every value, of basic type, whose dynamic dependencies he is interested in. The `ANSWER` function, for a program returning a list of integers `l`, where the user is interested in the dependencies of every element of the list would run like this,

```
val l  =  ...
val ANSWER = fn f =>  let fun F [] = ()
                            |  F (x::l) = f (makestring x) ; F l
                  in  (F l) end
```

In the annotated version of the program, we have an additional line in the program, where `ANSWER` is applied to another function which collects the dependencies of each of the values to which it is applied and returns the complete set of dependencies, as the slice.

The Structure of the Annotated Program

We will now illustrate the general structure of annotated programs. Consider, the SML program given in Figure 8.1. The annotated version of this program is given in Figure 8.2. The important points to note about the annotated program are:

```
local open Pervasives in

val EMPTY = Set_of_Labels.empty

val ID  = (fn z => z)

val INSERT = Set_of_Labels.insert

and  UNION = Set_of_Labels.union

and  SINGLETON = Set_of_Labels.singleton

val AUGMENT = (fn {1 = z10, 2 = z11} => (fn z12 =>
                 {1= z10 , 2= UNION z11 z12 } ))

and  AUGMENT_ONE = (fn {1 = z10, 2 = z11} => (fn z12 =>
                 {1= z10 , 2= INSERT z12 z11 } ))

val w = Pervasives.plus_opt2_int 8 9 (INSERT 3 (INSERT 2 (INSERT 1 EMPTY)))

val x = {1= 10, 2= (INSERT 4 EMPTY) }

val z = (fn y => fn z13 => AUGMENT  y (INSERT 5 z13))
        (AUGMENT w (INSERT 6 EMPTY) )
        (INSERT 7 EMPTY)

val ANSWER =
    {1= (fn f =>
          (fn z14 =>
           (let val {1 = z15, 2 = z16} = f
            in
            z15 (Pervasives.makestring_arg_int (AUGMENT z (INSERT 8 EMPTY)) 9)
            (INSERT 10 (UNION z14 z16))
            end))) ,
       2= (INSERT 11 EMPTY) }

val {1 = z17, 2 = z18} = ANSWER

val _ = z17 {1= Pervasives.output , 2= EMPTY} z18

end
```

Figure 8.2 : An annotated program generated by the implementation

- The SML Definition [48] defines the semantics of programs based on an initial environment called the Initial Dynamic Basis. Annotated versions of the functions, defined in this basis, are placed in the module `Pervasives`. It is locally opened for the execution of each annotated program.

- The final expression evaluated in the annotated program is the application of the annotated version of the function `ANSWER` to the function `(Pervasives.output)`. The function `(Pervasives.output)` stores the dependencies of every value, to which it is applied, in a global store from where it can be extracted when needed.

- The translation function on programs $[\![\ ]\!]$ is designed to include, in the slice, the label of every program node which contributes to the value returned. As an optimisation, the actual implementation no longer includes every such node label: only the labels which form the leaves of the parse tree are included in the slice. At the end of the computation the set of labels is closed to include the non-leaf labels into the set.

- The program in Figure 8.2 has had some cosmetic changes made to it to make it more readable:

  - The variable names `INSERT` , `AUGMENT` , `EMPTY` have been introduced in the program to enhance readability. In reality, the translator makes a pass over the program to isolate the largest variable name, based on lexical ordering. Every variable introduced by the translator is then formed by concatenating an integer to this string. The variables `z13` , `z14` are examples of this.

  - SML allows a program to define infix operators. The fixity information is needed and used only in the parse phase of the language implementation. Our translator needs to output a valid SML program that will pass through the parser. This means that we would need to preserve the fixity information of operators beyond the parse phase of the input program. We have decided not to do so. Instead, every variable v, in the input program, is replaced by `(op v)` in the output program. This ensures that output program parses, irrespective of the fixity of the operators.

- SML does not allow a program to define overloaded functions. But certain pre-defined overloaded operators are allowed as long as their exact type can be determined from the context of their use. But, we cannot use the annotated versions of these overloaded operators in an overloaded fashion. Hence, we need to preserve the type of every occurrence of an overloaded variable from the elaboration phase and then substitute an annotated version of the appropriate type.

  This is seen in the case of the operator `+` , or the function `makestring` in Figure 8.2. The annotated program has the operator `+` from the original program replaced by (`Pervasives.plus_opt2_int`). This function is an optimised annotated integer addition operator.

The Structure of the Log File

When the example annotated program in Figure 8.2 is executed by our implementation it generates a log file which contains execution statistics for the input and the annotated programs. The contents of the log in Figure 8.3 are mostly self-explanatory. The lines containing the size of the dynamic slice and the execution slice contain two entries respectively. As mentioned, in the previous subsection, the label collection strategy, in the implementation, only collects labels at the leaves of the parse tree. The first entry in the line gives the size of the slice, assuming such a label collection strategy. The second entry in the line gives the size of the set formed by upward closure of the slice on the parse tree. The line keyed by `Fraction` gives the corresponding ratios of the sizes of the dynamic slice to the execution slice.

A careful study of the log file will reveal a slight inconsistency, though. Every node of the input program in Figure 8.1 is executed. Hence, the second entry in the execution slice line should exactly equal the size of the input program, as stated in the first line of the log. This is seen not to be the case: the counts differ by 1. The reason for this small difference is the internal representation of expressions in SML, which are function applications. SML provides the programmer an ability to specify the fixity of function operators. For example,

```
p.sml:

Input Program Size(in Nodes):  35
Annotated Program Size(in Nodes):  195
Time to Annotate Input:  0.040000
Execution Time of Input Program:   0.050000
Execution Time of Annotated Program:  0.160000
Dynamic Slice Size(in Nodes):  10   30
Execution Slice Size(in Nodes):   11    34
Fraction:                          90.9090909090909%   88.2352941176471%
```

Figure 8.3 : A log file generated by the implementation

```
fun xor (x,y) = (x orelse y) andalso not(x andalso y)

infix xor

val e = true xor false
```

A parser for SML, after infix resolution, would internally represent the expression for
e as (op xor){1=true,2=false}. Notice, that all the nodes in the parse tree of this
expression cannot be mapped to unique positions in the input file. Hence, there will
be some discrepancy between a program counting the number of nodes in the parse tree
and another counting the number of nodes by the number of distinct textual locations.
Besides, the example of infix function operators, discussed above, SML provides a whole
set of derived forms [48] which are pre-processed into other language constructs of the core.
A derived form which arises frequently is illustrated below,

```
val w = {a= 20, b = 90}

val x = (#a w)
```

SML translates the expression (#a w) into (fn {a=x,...} => x) w. Once again, we have
a parse tree whose nodes which cannot be mapped to unique textual locations in the input
file.

This discrepancy is not very serious from our point of view. We are predominantly
interested in isolating nodes which make no contribution to the values specified in the
slicing criterion. This isolation can be adequately done wth a set composed of leaves of
the parse tree. Besides, for most of the program analysed by the author, the difference in

the value of the ratios is also very small.

The Structure of the Diff File

The diff file generated by a call to the function `final_answer` gives the labels of leaf-nodes of the parse-tree, of the input file, which form a part of the execution slice but are not a part of the dynamic slice, based on the slicing criterion defined in the input file. In the example program, in Figure 8.1, the value x is not required in the computation of the final value of z. Hence, the diff file contains the entry $(3, 8, 3, 10)$. This entry states that the third line of the input file between columns 8 and 10 makes no contribution to the answer. This is the label for the value 10 on the third line. An upward closure of this label would include the entire declaration `val x = 10`.

8.2.2 Annotating Patterns

The skeletal language LML, in Figure 4.1, used throughout this thesis, has a very primitive case statement, as compared to SML. SML allows arbitrarily nested patterns, wild cards for patterns and pattern rows. Let us review the rules for computation of slices for the case statement.

$$\frac{E \vdash M_1 \rightarrow -}{E \vdash l : \text{case}(M_1, C(x_1, \ldots, x_n) \Rightarrow M_2, y \Rightarrow M_3) \rightarrow -}$$

$$\begin{array}{c} LL, F, S_0, L_0 \cup \{l\} \Vdash M_1 \rightarrow C((V_1, L_1), \ldots, (V_n, L_n)), S_1, L \\ LL, F[x_1 \mapsto (V_1, L_1), \ldots, x_n \mapsto (V_n, L_n)], S_1, L \Vdash M_2 \rightarrow V, S_2, L' \\ \hline LL, F, S_0, L_0 \Vdash l : \text{case}(M_1, C(x_1, \ldots, x_n) \Rightarrow M_2, y \Rightarrow M_3) \rightarrow V, S_2, L' \end{array}$$

$$\begin{array}{c} LL, F, S_0, L_0 \cup \{l\} \Vdash M_1 \rightarrow C'((V_1, L_1), \ldots, (V_n, L_n)), S_1, L \quad C \neq C' \\ LL, F[y \mapsto C'((V_1, L_1), \ldots, (V_n, L_n))], S_1, L \Vdash M_3 \rightarrow V, S_2, L' \\ \hline LL, F, S_0, L_0 \Vdash l : \text{case}(M_1, C(x_1, \ldots, x_n) \Rightarrow M_2, y \Rightarrow M_3) \rightarrow V, S_2, L' \end{array}$$

Let a simple pattern be defined as a variable or a constructor applied to a tuple of variables. For a language, with simple patterns, M_2 and M_3, the expressions beneath the patterns, need to be evaluated under the set L, the dependency of the value to which M_1

113

evaluates. Hence, we have the following the annotation for a case statement,

$$[\![\, l : \text{ case}(M_1 ,\ \text{C}(x_1 ,\ \ldots ,\ x_n) \Rightarrow M_2 ,\ y \Rightarrow M_3)\,]\!]_{L,R} =$$

$$\text{let } L_0 = L \cup \{l\}$$

$$\text{in case } (\,[\![\, M_1\,]\!]_{L_0,R}\,) \text{ of } (\,\text{C}(x_1 ,\ \ldots ,\ x_n),\ L_1\,) => [\![\, M_2\,]\!]_{L_1,R}$$

$$|\,(\,y,\ L_1\,) => [\![\, M_3\,]\!]_{L_1,R}$$

$$\text{end}$$

For the purely functional part of LML, we have a theorem, Theorem 5.3.1, which states that the dynamic slice of the program exactly co-incides with the execution trace of a call-by-name/lazy evaluator. In trying to move from a language with simple patterns to a language with complex patterns this is the property we will seek to maintain. The call-by-name evaluation semantics, given in Table 5.3, need not make any assumptions about the order of evaluation in patterns because all patterns are simple. When complex patterns are introduced into the language, the call-by-name evaluator needs to specify the order of evaluation in patterns. Let us look at a sample program,

```
case x of
  A( B x , C y ) => f(x, y)
| A( x , y ) => g(x, y)
| _ => 20
```

Given this program, a call-by-name evaluator will first evaluate x to $A(w_1 ,\ w_2)$. Now, the evaluator needs to decide which of the closures, w_1 or w_2, is to be evaluated first and matched against the corresponding pattern. Call-by-name evaluators, with different choices, are semantically different. The strategy we have opted for is the traditional one: from left to right. Once the order of evaluation of subpatterns, for the call-by-name evaluator, has been decided we can compile complex patterns into simple patterns and use the annotation technique used before. The question, which might be raised here is, what so sacrosanct about Theorem 5.3.1? The answer is, it captures intuition. If x is bound to the value A(B 8, D 9), in the annotated program it will be bound to some value (A(((B (8,L4), L2) , (D (9,L5), L3)) , L1) , L0). Even though the value of x matches only the second pattern, this does not mean that the

computation `g(x, y)` is dependent only on (`union L0 L1`). It is dependent also on the fact that the first pattern failed to match. Hence, assuming left to right pattern matching within sub-patterns, the computation `g(x,y)` should be considered dependent on (`union (L0 (union L1 (union L2 L3)))`).

Since the translation to simple patterns is done only to obtain the set of dependencies with which to compute the body of the case statement, we break up the case statement, in the annotated program, into two parts. The first part, works on simple patterns and returns the set of dependencies required for the pattern match. The second part annotates the expression associated with each rule, in the case statement, assuming it depends on the set of labels returned from the first part. The algorithm, used in the implementation, for the compilation of complex patterns to simple patterns, has been take from Augustsson [11]. This is briefly elaborated in Table 8.1. The compilation of a list of complex patterns, $[p_1, \ldots p_n]$, begins with a call to,

$$\text{let val } Id = \text{fn } x \Rightarrow x$$

$$\text{val } L = \text{emptyset}$$

$$\text{in } \mathbf{C}[\![(x, \left\{ \begin{array}{c} [p_1] \\ \ldots \\ [p_n] \end{array} \right\})]\!]_{L,Id} \text{ end}$$

In a call to $\mathbf{C}[\![ ]\!]_{L,f}$, L is the name of the variable which stores the set of labels needed in the matching till now and f is a call-back function which starts pattern matching along another path, if the match fails along this particular path. The cases elaborated in Table 8.1 are explained below:

I. This stands for a successful termination of matching, at run time. We simply return the set of labels collected in the course of the match.

II. This case is actually an instance of case (3). When the head of all pattern lists, to be matched, are variables, we proceed to match the tail of all the pattern lists.

III. This is most general case: the head of a pattern list may either be a variable or a constructor. In this case, the list of pattern lists is partitioned into sublists which

consist of pattern lists whose heads are exclusively variables or exclusively construc-
tors. Table 8.1 shows the list of pattern lists getting partitioned into three sublists.
We then define two new default functions. One defining the path to follow if no
pattern list in the first sublist matches and the other defining the path to follow if
no pattern list in the second sublist matches.

The head of every pattern list, in the first sublist, is a constructor. Within this sublist,
we bring adjacent to each other pattern lists with the same constructor at their head.
For example, for every pattern list P_i, within the sublist, with the constructor C_1
at the head, we strip off the constructor and cons its arguments to the tail of the
pattern list, to form a new pattern list $P_{i'}$. Then we define a new simple pattern
for the constructor C_1, as a part of a case statement, collect its dependencies and
recursively match for the list of pattern lists $\{\, P_{i'} \,\}$. It is important to note that both
the default function and L have now changed. The collection of dependencies for
unary and n-ary constructors are different. Hence, we illustrate the case for both.

If the partition of pattern lists returns only one sublist, i.e. the head of every pattern
list begins with a constructor, then we will have no new default function to define.
All default function calls will use the input default function.

The description in Table 8.1 is very high-level and abbreviated. The actual implemen-
tation handles a lot of instances as special cases to prevent an explosion in code size.

It should be noticed that the label collection strategy is an implementation of *strict*
pattern matching [52]. We explain the concept through an example,

```
datatype TT = A of int
val  f = fn (A x) => 9
val z = f (A 2)
```

Slicing this program for the value of z returns the fact only the value 2 is superfluous. But,
intuitively, since the pattern (A x) is an exhaustive pattern of type TT and x is not used
in the body of the function, the entire expression (A 2) should be considered superfluous.
A lazy evaluator which does not evaluate its argument to match against a single pattern,
that is exhaustive, is said to implement *lazy* pattern matching. Currently our decision to
implement strict pattern matching is mainly for simplicity of implementation.

116

I. $\mathbf{C}[\![\,([\,],\{\,\})\,]\!]_{L,f} \;=\; L$

II. $\mathbf{C}[\![\,([x_1\,,\;x_2\ldots x_n]\,,\;\left\{\begin{array}{l}[v_1,p_{12},\ldots p_{1n}]\\ [v_2,p_{22},\ldots p_{2n}]\\ \ldots\\ [v_m,p_{m2},\ldots p_{mn}]\end{array}\right\}\,)\,]\!]_{L,f}\;=$

$$\mathbf{C}[\![\,([x_2\,,\;\ldots x_n]\,,\;\left\{\begin{array}{l}[p_{12}\,,\;\ldots p_{1n}]\\ [p_{22}\,,\;\ldots p_{2n}]\\ \ldots\\ [p_{m2}\,,\;\ldots p_{mn}]\end{array}\right\}\,)\,]\!]_{L,f}$$

III. $\mathbf{C}[\![\,([x_1\,,\;x_2\ldots x_n]\,,\;\left\{\begin{array}{l}[C_1(q_{11},\ldots,q_{1n1}),p_{12},\ldots p_{1n}]\\ [C_2\,q_{21},p_{22},\ldots p_{2n}]\\ \ldots\\ [v_l,p_{l,2},\ldots p_{l,n}]\\ \ldots\\ [v_{k-1},p_{k-1,2},\ldots p_{k-1,n}]\\ [C_k(q_{k1},\ldots,q_{kn_k}),p_{k2},\ldots p_{kn}]\\ \ldots\end{array}\right\}\,)\,]\!]_{L,f}\;=$

`let fun` $h\;L = \mathbf{C}[\![([x_1\,,\;x_2\ldots x_n],\left\{\begin{array}{l}[C_k(q_{k1},\ldots,q_{kn_k}),p_{k2},\ldots p_{kn}]\\ \ldots\end{array}\right\})\,]\!]_{L,f}$

$\qquad$ `fun` $g\;L = \mathbf{C}[\![\,([x_2\,,\;\ldots\,,\;x_n]\,,\;\left\{\begin{array}{l}[p_{l,2},\ldots p_{l,n}]\\ \ldots\\ [p_{k-1,2},\ldots p_{k-1,n}]\end{array}\right\}\,)\,]\!]_{L,h}$

`in`

`case` x_1 `of`

$\quad(C_1((y_1\,,\;\ldots\,,\;y_{1n1})\,,\;L_0)\,,\;L_1)$ `=> let val` L_2 `= union` L `(union` $L_0\;L_1)$

$\quad$ `in` $\mathbf{C}[\![\,([y_1\,,\;\ldots\,,\;y_{1n1}\,,\;x_2\ldots x_n]\,,\;\left\{\begin{array}{l}[q_{11},\ldots,q_{1n1},p_{12},\ldots p_{1n}]\\ \ldots\end{array}\right\}\,)\,]\!]_{L_2,g}$ `end`

$\mid (C_2\,y_1\,,\;L_0)$ `=> let val` L_1 `= union` $L\;L_0$

$\quad$ `in` $\mathbf{C}[\![\,([y_1\,,\;x_2\ldots x_n]\,,\;\left\{\begin{array}{l}[q_{21},p_{22},\ldots p_{2n}]\\ \ldots\end{array}\right\}\,)\,]\!]_{L_1,g}$ `end`

$\mid\;\ldots$

$\mid (\_,\;L_0)$ `=>  let val` L `= union` $L\;L_0$

$\qquad\qquad$ `in` $(g\;L)$ `end`

`end`

Table 8.1: Compiling Complex Patterns to Simple Patterns

8.2.3 Optimising the Program Slicer

The naive program annotator function, $[\![\ ]\!]$, as described in the beginning of this chapter, provides for poor runtime performance. The predominant reason for the high execution time of the annotated program is the large number of set unions and insertions being performed in the course of the computation. The entire suite of implemented optimisations is directed towards reducing the number of set unions and insertions being performed by the annotated program. We have already mentioned one important optimisation in the previous subsection: of the label insertions, introduced by the annotator function, the only ones that are retained are those which involve insertions of labels that are leaves of the parse tree. Insertion of a non-leaf label is superfluous: it can be included in the slice at the end of the computation, using the order of labels defined by the parse tree.

Function Application

Atomic Operators, particularly arithmetic operators, occur very frequently in programs. Hence, it is extremely important to weed out any superfluous computation in the annotated version of these operators. Consider, the integer addition operator (op +): int*int -> int. The type of the annotated operator $[\![$int*int -> int$]\!]$ equals ((int*L) * (int*L))*L -> L -> int*L where L = Set_of_Labels. Hence, the immediate annotated version of this operator is,

```
val plus_int0 = fn(((x1:int,L1),(x2,L2)),L3) => fn L0 =>
                    (x1 + x2, union L0 (union L1 (union L2 L3)))
```

But the operator + is rarely, if ever, applied to a term that is not an explicit pair. Integer expressions like (op +) (f x y) rarely occur and the function plus_int0 is needed only for handling such expressions. Whenever the operator + is applied to an explicit pair, it may be that either one or both components of the pair are integer constants or neither are. Based on these possibilities, we define four additional different integer operators:

```
val plus_int1 = fn x1:int => fn (x2,L2) => fn L3 => (x1 + x2, (union L2 L3))
```

```
val plus_int2 = fn(x1:int,L1) => fn x2 => fn L3 => (x1 + x2, (union L1 L3))
```

118

```
val plus_int3 = fn x1:int => fn x2 => fn L => (x1 + x2, L)


val plus_int4 = fn(x1:int,L1) => fn (x2,L2) => fn L3 =>
                                    (x1 + x2, union L1 (union L2 L3))
```

All these operators perform lesser number of set unions than the operator `plus_int0`. It should be noticed that all these operator have different types. Whenever the operator `+` appears in an input program, the implementation finds out which version to use and annotates the arguments accordingly. A similar optimisation could be performed for all binary functions which do not escape the scope of their declarations. For such functions, if we know all the call-sites, we can keep around multiple annotated versions of the function, using whichever one is appropriate.

In general, given a function taking in an argument of type $\tau_1 * \tau_2$, its annotated version takes in an argument of type $(\llbracket \tau_1 \rrbracket * \llbracket \tau_2 \rrbracket) * \mathrm{Set\_of\_Labels}$. The annotation for an explicit pair $l : (M_1, M_2)$, is given by $\llbracket l : (M_1, M_2 \rrbracket_L \equiv ((\llbracket M_1 \rrbracket_{L \cup \{l\}}, \llbracket M_2 \rrbracket_{L \cup \{l\}}), L \cup \{l\})$. Hence, the annotated version of the term $l_0 : M_0 (l : (M_1, M_2))$, is given by,

$$\llbracket l_0 : M_0 (l : (M_1, M_2)) \rrbracket_{L_0} \equiv \quad \mathrm{let} \quad L_1 = L \cup \{l_0\}$$
$$(f, L_2) = \llbracket M_0 \rrbracket_{L_1}$$
$$v = ((\llbracket M_1 \rrbracket_{L_1 \cup \{l\}}, \llbracket M_2 \rrbracket_{L_1 \cup \{l\}}), L_1 \cup \{l\})$$
$$\mathrm{in} \quad f \, v \, L_2 \; \mathrm{end}$$

It is to be noticed that the label set component of the value v is already used in the annotation of the components of the pair. Except for the label l, the label set component of v is also contained in the set of labels passed as an argument to the function f. Hence, we can change the annotation of the pair to $v = ((\llbracket M_1 \rrbracket_{L_1 \cup \{l\}}, \llbracket M_2 \rrbracket_{L_1 \cup \{l\}}), \{l\})$. Reducing the size of the label set component significantly reduces the cost of a set union.

A similar optimisation can be done when a function is applied to a variable. The standard annotation for $l_0 : M_0 (l : x)$ is given by,

$$\llbracket\, l_0 : M_0\,(l:x)\,\rrbracket_{L_0} \equiv \text{let} \quad L_1 = L \cup \{l_0\}$$
$$(f, L_2) = \llbracket\, M_0\,\rrbracket_{L_1}$$
$$(v, L_3) = x$$
$$x = (v, L_3 \cup L_1 \cup \{l\})$$
$$\text{in} \quad f\,x\,L_2 \;\;\text{end}$$

Clearly, the augmentation of the set component of x by L_1 is redundant. This is because within the body of f, x will be evaluated under L_2, which contains L_1. Hence, we need only have $x = (v, L_3 \cup \{l\})$. In general, function applications to values can accommodate this kind of optimisation.

A function of type $(\tau_1 \to \tau_2)$ get translated to a function of type $(\llbracket\,\tau_1\,\rrbracket \to \mathtt{Set\_of\_Labels} \to \llbracket\,\tau_2\,\rrbracket) * \mathtt{Set\_of\_Labels}$. For functions *values* which have *single* call-sites, which are *known*, the standard annotation does a lot of superfluous computation. The standard annotation for $l_0 : (l:\,\lambda x\,M)\,M_1$ would be given by,

$$\llbracket\, l_0 : (l:\,\lambda x\,M)\,M_1\,\rrbracket_L \;\; \equiv \;\; \text{let} \quad L_0 = L \cup \{\,l_0\,\}$$
$$(f\,,\,L_1) = (\lambda x\,\lambda L_1\,\llbracket\, M\,\rrbracket_{L_1}\,,\,L_0 \cup \{\,l\,\})$$
$$V = \llbracket\, M_1\,\rrbracket_{L_0}$$
$$\text{in}\;f\,V\,L_1\;\text{end}$$

Our implementation, which also collects labels only at the leaf of the parse tree, would annotate this as,

$$\llbracket\, l_0 : (l:\,\lambda x\,M)\,M_1\,\rrbracket_L \;\equiv\; (\lambda x\,\llbracket\, M\,\rrbracket_L)\,\llbracket\, M_1\,\rrbracket_L$$

This is correct only because the function is a value and its call-site is known.

The List Library

Lists are the most frequently used datatypes in SML and it is of utmost importance to optimize the annotated version of list functions. The datatype `list`, in the annotated program, is given by the following declaration:

```
datatype 'a list = nil
  | :: of (('a*Set_of_Labels)*('a list*Set_of_Labels))*Set_of_Labels
```

A value of type `int list`, in the source program, becomes a value of type $\llbracket\,\mathtt{int\,list}\,\rrbracket * \mathtt{Set\_of\_Labels}$ in the annotated program, where $\llbracket\,\mathtt{int\,list}\,\rrbracket$ is an instance

of the annotated datatype declaration for `list`. Just as for arithmetic operators being applied to tuples, every application of the cons constructor is optmised to make sure that no redundant information is computed.

Operations on lists, or for that matter, any recursive datatype, generate an enormity of set operations. The reason for this can be illustrated through an example. Suppose we want to pull out the tenth element of a list starting from the head. Since we have to walk down from the head of the list to the tenth element, this operation depends on the dependencies of all operations required to construct the list upto that point. Hence, operations involving traversal of recursive data structures usually require the unioning of all label sets required for the construction of the structure itself.

Let us look at a simple implementation of the append function for lists,

```
fun (a::rest) @ tl = a :: (rest @ tl)
  | nil @ tl = tl
```

Let us now take a look at a readable bare bones representation of the annotated version of the above program. We will use the constructor name `CONS` instead of `::` because we have stripped off irrelevant information. We will be using a curried form to avoid clutter.

```
fun  Append (CONS(a,rest),L1) tl L0 = let val L = union L0 L1
                                         in (CONS(a, Append rest tl L),L) end
| Append (NIL,L1) (tl as (ll,L2)) L0 = let val L = union L0 (union L1 L2)
                                         in (ll,L) end
```

Looking at this program, we see that we are constructing the union of the sets of dependencies required to reach every point in the first list and are placing this union in the corresponding position in the appended list. This is a lot of computation. To optimise the annotated version of append we use two critical pieces of information about the semantics of SML:

- The only way to access the nth element of a list is to destructure it through a sequence of n pattern matches. There is no way to preserve a pointer to the nth element and use it to access the value.

- The append function does not produce any side-effects and cannot generate any exceptions.

The two observations above tell us that we need not construct and carry around the set of dependencies required to reach a point in the list. This is because the append function returns no new values and raises no exceptions: it merely returns an augmented list. A bare bones presentation of the optimized version is as follows,

```
fun Append_Opt (CONS(a,rest),L1) tl LO =
      let fun Append (CONS(a,rest),L1) tl = (CONS(a, Append rest tl), L1)
            | Append (nil,L1) (tl as (ll,L2)) = (ll, union L1 L2)
      in (CONS(a, Append rest tl), union LO L1) end
| Append_Opt (NIL,L1) (tl as (ll,L2)) LO = let val L = union LO (union L1 L2)
                                           in (ll,L) end
```

This function performs exactly two set unions irrespective of the length of the first list: one at the beginning and one at the end of the first list. What is important to keep in mind, is the fact that the optimisation is correct only because the language is functional and there are no side-effects, within append.

The list function reverse is yet another function which produces no side-effects and is guaranteed not to raise an exception. Here is the standard implementation of reverse:

```
fun reverse l = let fun rev nil l = l
                      | rev (a::rest) l = rev rest (a::l)
                in (rev l nil) end
```

A bare bones annotated version of rev would run as follows,

```
fun rev (nil,L1) (l,L2) LO = (l, union LO (union L1 L2))
  | rev (CONS(a,rest),L1) l LO = let val L = union LO L1
                                 in rev rest (CONS(a,l),L) L end
```

Notice that the set of labels attached to the head of the reversed list is the union of the set of labels, attached to each point of the original list, and it contains the set of labels attached to any point in the reversed list. Since we cannot access a point in a list without

going through the head we may actually assign the set of labels attached to any point in the interior of the list to be the emptyset. An optimized version of `rev` would run as follows,

```
fun rev_opt (nil,L1) (l,L2) L0 = (l, union L0 (union L1 L2))
  | rev_opt (CONS(a,rest),L1) l L0 = rev rest (CONS(a,l),empty) (union L0 L1)
```

8.3   Applying Slicing to aid Program Development and Debugging

In this subsection we decided to apply slicing techniques to aid us in debugging and testing of a small application we wrote ourselves. The application chosen is a small toy interpreter for the λ-calculus. We chose this program because applications, like compilers, are typical examples where slicing can be of great use. If a program, which is a function from a base type to a base type, has a succinct algebraic specification, then the dynamic slice, of the value returned, is usually very close to the execution slice. For such programs, a difference, in the size of the dynamic slice and the execution slice, can usually be attributed to dead-code/superfluous computation in the program. For example, a program to compute the determinant of a matrix or a program performing a decomposition of a matrix should have its executable dynamic slice exactly co-inciding with its execution slice, unless encoding has been sloppy. Applications, like compilers, are general purpose programs meant to translate an arbitrary input from a given grammar. Hence, for a specific input, a lot of intermediate values make no contributions to the value output by the program.

We decided to implement the SECD machine [41] for the evaluation of terms in the simple λ-calculus with product types, constants of integer and boolean types and a collection of primitive operations on base types. The structure of the program is briefly outlined here:

- We define a datatype `term` which specifies the structure of λ-terms input to the program.

- We define a function `generate_code` which takes in value, of type `term`, and translates it into a sequence of instructions for the SECD machine. It checks that there

are no free variables in the input and removes bound variables by replacing them with corresponding de Bruijn [25] indices.

- The function `SECDEval` is the implementation of the SECD machine: it takes in the sequence of instructions supplied by `generate_code` and returns a value.

After completing the encoding of the application, we ran it on the term,

$$(fn\ x\ =>\ x)\ (4\ +\ 8)$$

The program returns by raising an exception, indicating that it cannot find the binding for the variable at run time. We apply the following slicing criterion on the program,

```
val ANSWER = fn f => (pretty_print
                        (SECDEval([],[],(generate_code 1 []   tt),EMPTY)) f )
                     handle _ => f "Uncaught Exception"
```

The term `tt` is bound is to the input term given above and `pretty_print` is a function for printing out values computed by the SECD machine. The log file of the slice computation is given in Figure 8.4. If the dynamic slice uses any data constructor then it also includes the corresponding datatype declarations. If we throw out the datatype declarations from both the execution and the dynamic slice then the size of the dynamic slice is found to be even more smaller than the size of the execution slice.

```
FinalExperiments/SECD/secd.sml:

Input Program Size(in Nodes):  1726
Annotated Program Size(in Nodes):  9636
Time to Annotate Input:  5.670000
Execution Time of Input Program:   0.760000
Execution Time of Annotated Program:  8.690000
Dynamic Slice Size(in Nodes):  129    407
Execution Slice Size(in Nodes):   160    446
Fraction:                        80.625%   91.2556053811659%
```

Figure 8.4 The log file for the execution of secd.sml

A study of the `diff` file allows us to zoom onto the location of the error immediately. In our initial implementation, we were not setting the value of the de Bruijn index for the

variable x properly: we were starting our counts from 1 instead of 0. The diff file points out that the following segments of the computation made no contribution to the raising of the exception:

- Generation of the code for the argument to the function and generation of the code for the function itself. Of course, the code for the body of the function forms a part of the dynamic slice.

- Evaluation of the argument, the evaluation of the function to a closure and the saving of the program state prior to the making of the function call are shown to make no contribution.

Typically, the testing of an interpreter is done by elaborating the actions which the interpreter performs for different program states. Then we use different input programs which reach each of those program states and observe the output. Let the interpreter be structured such that the actions performed by the interpreter for different program states are represented by different functions. A strategy which attempts to evaluate the coverage of a test suite by checkmarking the collection of such functions, which get executed in the testing process, is not adequate. This is because the execution of many of the functions return data structures as values and unless these values are used in the subsequent computation the function cannot be considered to have been tested. This is illustrated by performing a dynamic slice computation on the same input, (fn x => x) (4 + 8), after fixing the bug. The dynamic slice of the program is found to be 89.13% of the execution slice. Important points about nodes present in the execution slice but not in the dynamic slice are:

- The code constructing the closure for a function is found to make no contribution to the computation. This is as it should be: the function (fn x => x) has no free variables.

- Right before executing the body of a function, in a function call, the interpreter saves the current stack and the current environment. After the evaluating the body of the function it restores the stack and the environment. But for the input program given to the interpreter, in our experiment, there is no computation to be performed

after returning from the function call. Hence, our slicing technique tells us that the contents of stack and the environment, saved and restored, make no contribution to the computation. Thus, we have not tested the correctness of the implementation of the save and restore routines. Looking at it from another perspective, the dynamic slice points out that the saving and restoration of the local stack and environment, when there is no subsequent computation to be performed after the return from the current function call, is redundant. We must restructure our implementation to ensure that this redundant computation is not done: we need to eliminate *tail recursion*.

We now apply our interpreter to a term whose evaluation involves a non-trivial closure construction and computation after the return from a function call. The term input to the interpreter is,

```
(fn F => F (fn x => 3*x)) (fn f => fn x => x * (f (f x))) 4
```

The dynamic slice for the computation of this term is now 98.18% of the execution slice.

8.4 Slicing SML/NJ Compiler Benchmarks

SML/NJ uses a set of benchmark programs for studying the performance of the compiler. We will apply our program slicer on some of these programs to study the feasibility of our technique. A very through experimental analysis of dynamic slicing of C programs has been carried out in [64]. An important criteria used for selecting test cases in [64], which we will also be using, is

> To keep trace sizes reasonable, it was also necessary to select test cases that exercised a large number of different parts of the program while avoiding unusually long repeated executions of the expressions that contributed no additional information to the construction of slices.

Since functional programs allocate large amount of data dynamically the memory requirements for the execution of the annotated program is a particularly acute problem.

As mentioned previously, our implementation of the program slicer is two steps short of the whole of SML,

- Semantics for the slicing of modules and functors. The programs from the SML/NJ compiler benchmark, investigated here, do not involve serious application of functors. But most of these programs are broken into modules, scattered across different files. It took very little effort to concatenate files together and generate a core SML program for the trials.

- Annotating polymorphic equality. Quite a few of the programs, investigated, had polymorphic functions with equality types. Since our program slicer can handle equality only on basic types, we needed to instantiate such polymorphic functions with monomorphic types and provide for equality functions on non-basic datatypes. Quite surprisingly, in all of the cases we handled, we merely needed to write a couple of equality functions, usually for `list`, and instantiate the polymorphic function to a couple of datatypes.

8.4.1 The Boyer-Moore Theorem Prover

This is a small program implementing proof search techniques, for propositional logic, developed by Boyer and Moore [16]. A brief outline of the structure of the program is given below:

- A datatype `term` is defined: it is either a variable or a proposition built out of propositional connectives and the if-then-else function.

- Various functions for manipulation of terms are implemented:

  - An equality function on terms and term lists.

  - An environment, binding variables to terms, and associated functions for the manipulation of environments.

  - A substitution function for replacing variables with terms.

  - A unification function on terms.

127

- The theorem prover has a main function, `tautp`, which returns true for provable tautologies. The theorem prover works by term rewriting. Using the axioms for the propositional connectives, it completely rewrites the term into a proposition built solely out of the if-then-else function, variables and the constants `"true"` and `"false"`. This term is then simplified, using an axiom for the if-then-else function, to ensure that the first argument to the if-then-else function is always a variable or a constant. Such a term is now interpreted in the obvious way: if the first argument to the if-then-else function evaluates to `"true"` we evaluate the second argument or if it evaluates to `"false"` we evaluate the third argument. If it evaluates to neither `"true"` nor `"false"` the law of 'excluded middle' is applied.

- The slicing criterion is given by,

```
val ANSWER = fn f => if tautp (apply_subst subst term1)
                        then f "Proved!" else f "Cannot prove!"
```
where,
  - The variable `term1` is set to the formula $(A \Rightarrow B) \wedge (B \Rightarrow C) \Rightarrow (A \Rightarrow C)$.
  - The function call (`apply_subst subst term1`) applies substitutions to remove function symbols from `term1`.

The log file generated by the execution of the annotated version of the program is given in Figure 8.5. Important points about the nodes present in the execution slice but not in

```
FinalExperiments/boyer/Boyer.sml:

Input Program Size(in Nodes):  1817
Annotated Program Size(in Nodes):  10508
Time to Annotate Input:  6.620000
Execution Time of Input Program:   0.830000
Execution Time of Annotated Program:  21.280000
Dynamic Slice Size(in Nodes):  433    833
Execution Slice Size(in Nodes):   527    923
Fraction:                       82.1631878557875%   90.249187432286%
```

Figure 8.5 The log file for the execution of Boyer.sml

the dynamic slice are:

128

- The main function `tautp` applies the rewriting rules, defined by the axioms, to transform the input term into a proposition built solely out of the if-then-else function, variables and constants. The program stores the axioms in an associative list consisting of connective name and axiom list pairs. Hence, connective-axiom list pairs which were not used in the rewriting of the input term are not contained in the dynamic slice but contained in the execution slice. This is because they were executed during the construction of the associative list.

  An axiom for a connective consists of a pair of terms: the second component gives the term the first component rewrites to. Hence, if the first component of the axiom is present in the dynamic slice, but not the second component, we know that the program attempted to use this axiom, for rewriting, but the unification failed.

- Whenever the program attempts to apply a substitution function to a variable but finds that the variable name is not present in the domain of the substitution function it raises the exception, `failure "unbound"`. The slicing program discovers that the string argument to the exception constructor is never used subsequently.

8.4.2 Knuth-Bendix Completion

Given a signature[1] Σ_0, and a collection of equations E, on terms over Σ_0, the terms over Σ_0 can be partitioned by the smallest congruence relation built by extending the relation defined by E. The *word problem* is the problem of deciding whether two words belong to the same congruence class, i.e are they provably equal, equationally? The Knuth-Bendix Completion [39] procedure provides a solution to a subclass of the word problem. The idea of the solution runs as follows: if the equations can be oriented to form rewrite rules, in a *terminating* rewriting system, then two terms can be shown to be equal if they reduce to the same normal form. This strategy is valid only if every term rewrites to a unique normal form. The completion algorithm provides a technique to generate new equations/rewrite rules to make the system confluent, while preserving the associated equational theory.

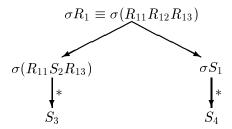The main function of the program, `kb_completion`, can be briefly outlined as follows:

- For every pair of rewrite rules, $R_1 \rightarrow S_1$ and $R_2 \rightarrow S_2$, `kb_completion` collects the

---

[1]Refer to Section 3.1 for formal definitions on term rewriting systems

associated list of *critical pairs*.

Let $R_1 \equiv R_{11}R_{12}R_{13}$, where R_{12} is a non-variable subterm.

Let σ be the most general unifier [56] for R_{12} and R_2, i.e. $\sigma(R_{12}) \equiv \sigma(R_2)$.

$\sigma(R_{11}S_2R_{13})$ and σS_1 are defined to be *critical pairs*.

$$\sigma R_1 \equiv \sigma(R_{11}R_{12}R_{13})$$

$$\sigma(R_{11}S_2R_{13}) \qquad\qquad \sigma S_1$$

$$\downarrow * \qquad\qquad\qquad\qquad \downarrow *$$

$$S_3 \qquad\qquad\qquad\qquad S_4$$

The function `kb_completion` then attempts to normalise both the terms in the critical pair to normal forms, in the leftmost outermost style. If the normal forms S_3 and S_4 are not identical then a new rewrite rule needs to be constructed from the pair. The orientation of the rewrite rule is based on a weight function, on terms, which is taken in by `kb_completion` as input. The new rewrite rule, involving S_3 and S_4, is oriented from the heavier to the lighter, in weight. If the weights of S_3 and S_4 are identical, but the terms are not, it adds $(S_3\,,\,S_4)$ to a list of term pairs called *failures*.

- Whenever it adds a new rewrite rule, the entire new set of rewrite rules E is checked to make sure that it contains only irreducible terms. For any rewrite rule $R \to S$, it finds normal forms R_0 and S_0, such that $R \overset{*}{\to} R_0$ and $S \overset{*}{\to} S_0$ with respect to rewrite rules $E - \{(R \to S)\}$. If the normal forms R_0 and S_0 are not identical then it introduces, into $E - \{(R \to S)\}$, the rewrite rule, involving R_0 and S_0, oriented from the heavier to the lighter, in weight. If the weights of R_0 and S_0 are identical, but the terms are not, it it adds $(R_0\,,\,S_0)$ to a list of term pairs called *failures*.

- Whenever it adds a new rewrite rule, it checks whether any term pair, $(R\,,\,S)$, in the list *failures* can now be normalised, with the new set of rewrite rules, to $(R_0\,,\,S_0)$, where $R \overset{*}{\to} R_0$ and $S \overset{*}{\to} S_0$. The following are the possible cases, regarding R_0 and S_0,

  - R_0 and S_0 are identical terms. The pair $(R\,,\,S)$ is removed from *failure* and the rest of *failure* is processed.

- R_0 and S_0 are not identical, but have the same weight. The pair (R_0, S_0) is introduced into *failure*, in the place of (R, S), and the rest of the list is processed.

- R_0 and S_0 are not identical but have different weights. The appropriately oriented rewrite rule is added to the existing set of rewrite rules and the rest of *failure* is processed.

The experiment we choose to conduct is to feed the program with the set of rewrite rules [2] given in (a), below. The program completes these rules to output the rewrite rules given in (b), below.

$$(Bx) * (Cy) \rightarrow Ex$$
$$B(Cx) \rightarrow Dx$$
$$Bx \rightarrow Dx$$

$$(Dx) * (Cy) \rightarrow Ex$$
$$D(Cx) \rightarrow Dx$$
$$E(Cx) \rightarrow Ex$$
$$Bx \rightarrow Dx$$

(a) Rewrite Rules Input   (b) Rewrite Rules Output

The slicing criterion in the input program is,

```
 val ANSWER = fn f => pretty_rules (kb_complete greater [] Group_rules) f
```

where, `greater` is the order function on terms, based on a weight function, and `Group_rules` is bound to the set of rewrite rules in (a).

In this experiment we are interested in the dependency associated with every rewrite rule output by the program. Hence, we call the pretty print function `pretty_rules` with the argument `f`, where `f` is the parameter of the `ANSWER` function. Since the input rules have been chosen to exercise the major cases of the function, we are interested in program coverage: sections of the program executed but making no contribution to the final answer. Such sections, though executed, can be claimed to have been left untested by the input data.

The log file generated by the execution of the annotated version of the program is given in Figure 8.6.

---

[2]The set of rewrite rules are chosen to be small but at the same time structured to ensure that the first two cases of the algorithm are exercised

```
FinalExperiments/knuth-bendix/knuth-bendix.sml:

Input Program Size(in Nodes):  4113
Annotated Program Size(in Nodes):  23594
Time to Annotate Input:  26.650000
Execution Time of Input Program:   2.720000
Execution Time of Annotated Program:  39.290000
Dynamic Slice Size(in Nodes): 737    1764
Execution Slice Size(in Nodes):   769    1800
Fraction:                       95.8387516254876%   98.0%
```

Figure 8.6 The log file for the execution of knuth-bendix.sml

The dynamic slice, in this experiment, is almost identical to the execution slice and provides little useful information over what is available from the execution slice. A study of the code reveals why. Most of the code is tail recursive and a lot of it is structured as follows:

- Try to perform an action, e.g. unify, match, rewrite, substitute. The actions return values if they succeed and raise exceptions if they fail.

- If successful then use the returned value in the subsequent computation.

- If failed then try performing another action.

In programs, with such structure, the dependencies associated with the performing of the action get passed on to the rest of the program, irrespective of whether the action succeeded or failed: if the action succeeded the value is used in the subsequent computation and if the action failed the rest of computation is control dependent on the action. For programs, with such structure, useful information can be obtained through slicing techniques only if we can achieve a partition between 'data' and 'control' dependencies. We investigate this in the next subsection.

Slices can serve as a guide to program comprehension and an aide to optimising programs. We will illustrate this with an example from our experiment. Terms, in this program, can either be variables or operators applied to a list of terms. They are given by the following datatype declaration,

```
datatype term = Var of int | Term of string * term list
```

The standard way of defining the location of a subterm within a term is with a list of integers[3]. For example, in `Term("*", [ (Var 3),Term("sin",[(Var 4)])])` the location of `(Var 4)` would be given by the list `[2,1]`. In Figure 8.7 we present the code, from the program, for the replacement of a subterm of a term `M`, at the location `u`, with another term `N`. The dynamic slice, from our experiment, points out that in the application of `f` to `h`, in the body of the function `change_rec`, the argument `h` is not used in the computation, but the value returned (f h :: t) is. Studying the code shows that this is always going to be the case whenever `u` is of length 1. This suggests that instead of going through superfluous function applications, for a very commonly occurring case, we should add an additional pattern to the function `reprec` to handle the case of the argument `u` having length 1, i.e. of the form (`n::nil`).

```
fun change f =
  let fun change_rec (h::t) n =  if n=1 then f h ::t
                                 else h ::  change_rec t (n-1)
        | change_rec _ _ = failwith "change"
  in change_rec
  end

fun replace M u N =
  let fun reprec (_, []) = N
        | reprec (Term(oper,sons), (n::u)) =
            Term(oper, change (fn P => reprec(P,u)) sons n)
        | reprec _ = failwith "replace"
  in reprec(M,u)
  end
```

Figure 8.7

Explicit Control Dependency

We decided to investigate as to what would be the dynamic slice of each of the four rewrite rules, output by the program, considered individually. To do this, we change the slicing criterion, at the end of the program, to the one given below. The variable `i` is assigned a

---

[3]Refer to Section 3.1 for a more formal exposition

value in [0 ... 3] depending on the rewrite rule we are slicing for.

```
val Complete_rules = kb_complete greater [] Group_rules
val ANSWER = fn f => pretty_rule f (nth (Complete_rules , i) )
```

We found that the dynamic slices associated with each of the four slicing criteria were identical. This is because the program ensures that, at every stage in the computation, the output list of rewrite rules consists only of irreducible terms: every rewrite rule, in the output list, is not reducible with respect to the other rewrite rules in the list. Hence, every rewrite rule is dependent on the dependency of every other rule. Hence, the computation of an executable dynamic slice is incapable of providing us with the following information: Given a rewrite rule, in the output list, what is the set of rewrite rules, in the input list, from which it is derived.

As explained, in the previous paragraph, every rewrite rule, in the output list, is dependent on every other rule in the output list. But this dependency is a 'control' dependency: once a rewrite rule is created a filter function, returning a boolean value, decides whether it can be retained in the list. The program is filled with examples of this kind of 'control' dependency.

In a higher-order programming language 'control' dependencies cannot be explicitly distinguished from 'data' dependencies. This is because instead of choosing a specific path of computation based on an if-then-else/case expression, we can have a computation return a projection function which is applied to a pair of expressions. As shown in [12], we can define the entire set of recursively enumerable functions, on Church numerals in pure lambda calculus. Thus, instead of attempting to formulate concepts of 'data' and 'control' dependencies for higher-order programs, we decided to apply the concept of control dependency used in first-order programs. Formal definitions of control dependency, in first-order programs with arbitrary gotos, can be formulated [18] in terms of post-dominators on the control flow graph. But in a simple first-order program, whenever a specific path of computation is chosen, based on the value of an expression as in an if-then-else/case statement, subsequent computations, within scope, are be said to be *control dependent* on the dependencies of the value of this expression. For higher-order programs, we will refer to this kind of a dependency as an *explicit control dependency*. We will not formulate this

concept rigorously: we simply want to look into what happens to slices, in our experiment, if we throw out explicit control dependencies. What we mean here is that, whenever we have function arguments which are patterns, such that the body of the function does not explicitly use the value of the input argument, then the dependency of the input argument is discarded. The if-then-else/case expressions in SML are derived forms of functions with patterns. Consider the set of functions shown below,

```
( fn true => M1 | false => M2 )
( fn (true,L1) => fn L0 => ⟦M1⟧_{L0 ∪ L1} | (false,L1) => fn L0 => ⟦M2⟧_{L0 ∪ L1} )
( fn (true,_) => fn L0 => ⟦M1⟧_{L0} | (false,_) => fn L0 => ⟦M1⟧_{L0} )
```

The first function is a standard if-then-else expression in SML. The second function is its standard annotation. It is to be noticed that the value returned by this function depends on the dependency of the boolean value. The third function is the annotation we are going to use for our investigation in this subsection: it ignores the dependency of the boolean value.

We extend this idea to all patterns: unless the function body explicitly uses a value from the pattern, the pattern will be considered to be a path selector and its dependencies will be ignored. The details of this extension are given in Table 8.2. The following two function are defined in Table 8.2,

- $\mathcal{P}⟦\ ⟧$, which takes in a pattern and a list of dependencies and returns an annotated pattern and a list of variables and their associated list of dependencies.

- $\mathcal{PE}⟦\ ⟧$, which takes in a pair consisting of a pattern and an expression as argument. It annotates the pattern using $\mathcal{P}⟦\ ⟧$ and then augments each variable by the list of its dependencies, as returned by the function call to $\mathcal{P}⟦\ ⟧$. It then annotates the expression using L0, completely ignoring the dependencies of the pattern.

Patterns are no longer compiled to simple patterns to capture the complete set of dependencies required to match a specific pattern out of a list of patterns. Unless the expression uses a variable from the pattern the dependencies associated with the pattern are discarded. Even when the expression uses a variable from the pattern the only set of dependencies included are those in the path from the root of the pattern tree to that specific

135

variable at the leaf. Exception handlers, which are also composed of pattern-expression pairs, are also annotated similarly.

We implemented this annotation for patterns and sliced the Knuth-Bendix Completion program, with the input from the previous experiment, and a slicing criteria composed only of the rewrite rule $D(Cx) \rightarrow Dx$. The rewrite rule $(Bx) * (Cy) \rightarrow Ex$, present in the input, is now no longer a part of the slice. The aim of the effort was to see whether we could partition dependencies into data and control dependencies, in the way Korel and Laski [40] did for first-order imperative programs, and see whether we could capture data dependencies between the input and the output. This seems to be possible. But we have not developed any theory nor any extensional characterisation of the slice we are computing. Hence, we leave this merely as an observation, for now, and a topic to be pursued, for the future.

This subsection discussed a refinement of the standard executable dynamic slice in which we ignore *all* explicit control dependencies. A strategy in which we allow the user to specify which of the case expressions, in the program, should ignore explicit control dependencies might have some applications. A standard rewrite rule which causes completion systems to fail is the commutative rule: mbox$x * y \rightarrow y * x$. Adding this rewrite rule to the set of input rules causes the completion program to exit with an exception. Computing the standard dynamic slice does not, as discussed, capture a strict subset of the input rules responsible for causing the exception. Ignoring all explicit control dependencies provides no information either since the program exited with an exception.

8.5   Fundamental Limitations and Proposals

On average, the annotated version of a program takes a factor of 15 more time to execute than the unannotated version. We feel a suite of optimisations of the standard of the SML/NJ compiler [9] would be be able to reduce the time to a more realistic factor of 10. We consider a factor of 10 to be realistic because a study on the dynamic slicing of C programs in [64] shows an average overhead of around 7.5. Programs in C have a static control flow graph and have a lot of data statically allocated. Unlike C programs, the core data structures of SML programs are recursive datatypes. Recursive datatypes consist of

$$\mathcal{P}[\![\_]\!]_l \ = \ (\_, [\,]\,)$$

$$\mathcal{P}[\![\,x\,]\!]_l \ = \ (\,x\,, [(x\,,\,l)]\,)$$

$$\mathcal{P}[\![\,C_0\,]\!]_l \ = \ ((C_0\,,\,\_)\,, [\,]\,)$$

$$\mathcal{P}[\![\,C(P_1\,,\,\dots P_n)\,]\!]_l \ = \quad let \quad (P_1, l_1) \ = \ \mathcal{P}[\![\,P_1\,]\!]_{L::l}$$

$$\dots$$

$$(P_n, l_n) \ = \ \mathcal{P}[\![\,P_n\,]\!]_{L::l}$$

$$in \quad (\,(\,C(P_1\,,\,\dots\,,\,P_n\,)\,,\,L\,)\,,\,l_1\,@\,\dots\,l_n\,)\ end$$

$$\mathcal{PE}[\![\,(\,P\,,\,M\,)\,]\!] \ =$$

$$let \quad (\,P\,,\,l\,) \ = \ \mathcal{P}[\![\,P\,]\!]_{[]}$$

$$[\,(\,x1\,,\,[L11\,,\,\dots\,,\,L1n1]\,)\,,\,(\,x2\,,\,[L21\,,\,\dots\,,\,L2n2]\,)\,,\,\dots] \ = \ l$$

$$in \quad ($$

```
      P,  fn L0 =>
                  let  fun Augment (x,L0) L1 = (x , union L0 L1)
                        val x1 = Augment x1 (union L0 (union L11...L1n1))
                        val x2 = Augment x2 (union L0 (union L21...L2n2))
                        ...
```

$$in \quad [\![\,M\,]\!]_{L0}\ \text{\tt end}$$

$$)$$

$$end$$

Table 8.2 : Compiling Patterns to Ignore Explicit Control Dependencies

dynamically allocated memory which are linked through pointers. Hence, accessing the tailend of such structures is dependent on very large sets.

There are two important areas where optimisations will provide a significant boost to the performance of the annotated program:

- Optimisation of the annotation of functions and function calls. At the end of the section, discussing the implemented optimisations associated with function applications, we showed how to optimise function calls, of the form (fn x => ...)M, to eliminate a lot of superfluous computation at run time. Most SML programs are filled with expressions of the form,

```
                    let fun f x = ...

                    in ...  (f M) ...  end
```

If the only occurrences of f, in the scope of this declaration, are function applications to arguments then the optimisation implemented for calls for the form

137

`(fn x => ...)M` can also be applied here. Another frequently occurring expression is of the form,

```
let fun f x1 ...  xn = ...
in ...  (f M1 ...  Mn) ...  end
```

If the only occurrences of `f`, in the scope of this declaration, are function applications to n arguments then the function `f` can be transformed to an uncurried form taking in a single argument which is an n-tuple. We can then apply the optimisation discussed for the previous case. The savings associated with uncurrying will be huge. This is because for every application in the source program we have two applications in the annotated program: one involving argument and the other involving the set of dependencies associated with the call-site.

- Optimisation of the annotation of patterns. The translation of complex patterns to simple patterns, defined in Table 8.1, is a naive syntax-directed approach. Our implementation gives a special treatment to patterns of the type `boolean` and other datatypes with only nullary constructor. But a lot more could be done here, especially regarding the handling of patterns which are exhaustive.

Let `g` be the annotated version, of a SML function `f`, that is generated by our program. If `f` is small enough, with no more than simple patterns, we can, with some effort, write by hand a SML function `h` that performs the same computation as `g`. Since we, as individuals, have an understanding of the denotation of `f` and the concept of slicing the hand-written function `h` will probably be more efficient than the function `g`, which is generated through a syntax-directed approach. Whenever a recursive datatype is defined, in a program, the programmer also defines a small set of simple functions for traversal and manipulation of these data structures. Such functions, particularly the ones for the traversal of the data structure, usually end up becoming the most used functions in the program. Hence, allowing the user to provide functions to supplant annotated versions, generated from the source program, might be a good idea. The benefits can be very positive as has been discussed in the case of functions for manipulations of lists. These functions have been hand written and not generated from a source program. There is, of course, the obvious down side: the user supplied function may be semantically incorrect.

Large linked structures arise very frequently in SML programs. Consider, for example, the Boyer-Moore theorem proving(BMTP) program investigated as a part of the case study. The program operates by rewriting an input proposition into a simple if-then-else expression. All propositions are represented as tree structures. The rewrite rules are such that it causes a proposition, with a couple of nested implications in its antecedent, to blow up into an expression, which is exponential in the size of its input. Similarly, the Knuth-Bendix completion(KBC) program when attempting to complete a set of 3 rewrite rules, defining the axioms for groups [39], generates 41 rules in the intermediate computation. If each unit of data in the memory is now tagged, by the set of its dynamic dependencies, we have just increased our memory usage by a big factor. This points out that the primary limitation of slicing techniques is space. The annotated version of a program, which is big in size and also dynamically allocates huge amounts of data, very quickly runs out of memory. The annotated versions of both BMTP and KBC, for the kind of inputs discussed in this paragraph, quickly stagnate in their computation for lack of memory: we have 800Kb of data getting paged in and out every second.

The question is, what are the possible solutions to this problem? The most pragmatic solution is to make a careful decision about the labels which are to be collected in course of the computation. Currently, every node in the parse tree is assigned a unique label and the computation of the dynamic slice collects labels at the leaves of the parse tree. Depending on the application, we have a few options as to the labelling technique to use:

- A debugging application might find it sufficient to isolate the lines in the program which made a contribution to the value computed at a breakpoint. Hence, all nodes in the parse tree coming from the same line can be assigned the same label. For the KBC program, it cuts down the number of labels, at the leaf, by almost a factor of 10.

- A program developer might only be interested in detailed information about certain sections of the program. Every node in the parse tree, belonging to those sections, needs to be assigned a unique label. The structure of labels for the rest of the program is left dependent on the granularity of information we desire. All nodes in the rest of the program may assigned be the same label or we may, for instance, assign unique labels to functions and have all nodes within a function have the same label as the

function.

The parse tree of KBC has 4113 nodes and has 984 leaves. Thus the size of a set of dependencies that can potentially be associated with any unit of data, in this program, is 984. This number is way too big, especially, for a program dynamically allocating huge amounts of data. If the size of the set of labels to be manipulated can be traded for the amount of detail, using the techniques discussed above, to a couple of hundred then, using a bit-mapped representation for sets, we can limit the space usage of the annotated program to be no more than a factor of 5 of the space usage of the source.

8.6   Conclusion

We performed a limited number of experiments, involving the dynamic slicing of SML programs, using the tool we developed. This experience has provided us with a feel for the kind of higher-order programs where slicing can be very helpful and the kind of programs where slicing is of little use.

Typically programs performing numerical computations will have dynamic slices which are very close, in size, to the execution slice. Programs, like compiler benchmarks, are not very strongly data driven computations. What we mean is that program fragments executed for different inputs are almost identical: the number of iterations made are usually different. Sorting is a typical example of such a program. Dynamic slicing of such programs rarely generates any useful information over what is available from the execution slice.

A program whose computation is strongly data driven, i.e. control flow is very largely dependent on the value of the input data, usually has dynamic slices which are much smaller when compared to the execution slice. An interpreter for a programming language is a typical example of this. A slicing technique provides us with the ability to investigate which of the values constructed, at run-time, get used in the subsequent computation. This is particularly relevant for values of non-basic types because not all the components of a such value make a contribution to the final answer. This is the kind of information which is unavailable from execution slices but a great help in investigating coverage. This is also the kind of information that points out optimisations which can be put in place to eliminate redundant computation in special cases.

Chapter 9

Conclusion

In this chapter we summarize the central ideas of the thesis and discuss the viability of dynamic slicing as a program analysis technique and as a development and debugging aid for software written in higher-order languages. We outline the current status of this work and highlight some of the important areas of future work.

Dynamic Slicing of first-order programs has been around as a semi-formal concept since Weiser's seminal work [65] in his thesis in 1979. A formal definition of dynamic slices for first-order programs, based on denotational semantics, was formulated as late as 1991 by Venkatesh [63]. Our contribution lies in the presentation of a formal definition for program slices, of higher-order programs, based on operational semantics. We have formulated an algorithm, for the computation of dynamic slices, as a natural deduction semantics. This approach to the formulation of an algorithm, for the computation of slices, provides us with the ability to state a succinct correctness theorem and present a proof thereof. For purely functional programs, we show that minimum dynamic slices exist and co-incide with the set of terms that get executed under a lazy/call-by-name evaluator.

We discover that, with respect to the computation of executable dynamic slices, the move from purely functional programs to programs with imperative features, like assignments and exceptions, is a non-trivial jump. For higher-order imperative programs minimum dynamic slices no longer exist. The intuitive extension of the natural deduction semantics for the computation of dynamic slices, of purely functional programs, fall short

of computing an executable dynamic slice. We show that augmenting the natural deduction semantics, with a closure relation, is capable of computing executable dynamic slices for programs with both assignments and exceptions.

Computation of static approximations to executable dynamic slices is a technique to analyse a program for dead code. This is because it captures the two loosely-coupled notions constituting the concept of dead code: code that is never going to be executed and code that is going to be executed but will make no contribution to the final answer. For purely functional programs, we have developed the concept of a demand-driven set-based analysis. We provide a set-based semantics, for purely functional programs, that incorporates the concept of demand. This semantics helps us provide an extensional characterisation and a correctness proof of the algorithm for the computation of a static approximation to the dynamic slice of a program. Our research into set-based analysis has uncovered some foundational flaws in the previous research in this area [14].

Our formulation of the algorithm for program slicing, as a natural deduction semantics, provides us with another important benefit: a simple and correct implementation. Given a program, whose dynamic slice needs to be computed, we generate an annotated version of the program which collects the relevant information during its execution. We provide a simple correctness proof of the program annotation technique. We have built an implementation that can annotate any core SML program which does not perform equality operations on non-basic types.

We performed three small case studies to investigate the utility and limitations of slicing techniques in higher-order languages. While reading the conclusions in this paragraph it is be kept in mind that our slicing criterion was always the value returned by the computation and never any intermediate value in a computation. For such slicing criteria, we feel that slicing techniques are particularly useful for investigating coverage of test suites. Whenever an application constructs values of non-basic types, at run time, dynamic slicing can tell us which components of the value make a contribution to the subsequent computation. For the kind of slicing criteria mentioned, dynamic slices are around 80% of the size of the corresponding execution slice. In programs written in heavily imperative fashion, particularly, with lots of exceptions, the dynamic slice is very close to the execution slice and usually provides little relevant information over what is available from the execution

slice. In a lot of instances a study of the dynamic slice provides us with suggestions to restructure the program to get rid of redundant computations.

The fundamental limitation of slicing techniques developed in this thesis appears to be space. Every value in a computation is tagged by the set of its dependencies. Hence, programs which dynamically allocate large amounts of data quickly run out of memory. We have some proposals for trading space usage with the granularity of information obtained from a slice computation. Further research needs to be done to find out whether these proposals substantially reduce memory usuage.

The most immediate line of research which needs to be pursued is the extension of slicing techniques to modules and functors. Researchers, investigating slicing of first-order programs, have pointed out the utility of slicing techniques in comprehension and development of large software systems. The code for large software systems, implemented in SML, is usually heavily functorised. Formulating the concepts and algorithms for dynamic slicing, in the presence of modules and functors, would allow us to investigate production quality software like the SML/NJ compiler. A brute force extension, into the realm of functors, which deals only with value bindings and completely ignores computation on types, performed during a functor application, is immediately feasible. But an executable dynamic slice, which type checks and takes into account the generative nature of datatype declarations, in functor applications, will need substantial effort.

Another line of research which holds great prospects is the incorporation of our program slicer into a debugger. This would be along the lines of the SPYDER tool [4]. Whenever a program stops at a given break point we should be able to query the dynamic slice associated with the value of any variable in scope. Issues associated with the realising of this goal are more technical than foundational. Currently, if we place a breakpoint, in the corresponding position, in the annotated program and apply the current value of the closure function, at the breakpoint, to any value we can obtain its dynamic slice. The technical complexity is about textual mappings between the source program and its annotated version and the handling of exceptions.

A major challenge, which lies ahead of us, is the generation of annotated programs which perform extremely limited number of set unions. Isolating more and more special cases and treating them separately will substantially decrease execution time. Semantic

analysis which can isolate functions which do not escape their scope, or expressions whose evaluations do not raise exceptions or involve assignments, can also help reduce execution time significantly. Currently our strategy has been to tag every value with its dependency. It is worth exploring how much of this information can be reconstructed at the end of the computation with information dumped into flat files at strategic points in the execution.

The core of this thesis has been dedicated to laying the foundations of the concepts and algorithms associated with the dynamic slicing of higher-order programs. We have an implementation to show that the concepts developed here are feasible and useful. We believe that following up the research avenues indicated here would lead to a very practical and useful tool for analysing higher-order programs.

Bibliography

[1] M. Abadi, L. Cardelli, P.-L. Curien, and J.-J. Lévy. Explicit substitutions. *Journal of Functional Programming*, 1(4):375–416, 1991.

[2] M. Abadi, B. Lampson, and J.-J. Lévy. Analysis and caching of dependencies. In *Proceedings of the 1996 ACM SIGPLAN International Conference on Functional Programming*, pages 83–92, 1996.

[3] L. Aceto, B. Bloom, and F. Vaandrager. Turning SOS rules into equations. In *Proceedings of the IEEE Symposium on Logic in Computer Science*, 1992.

[4] H. Agrawal, R. DeMillo, and E. Spafford. Debugging with dynamic slicing and backtracking. *Software-Practice and Experience*, 23(6):589–616, 1993.

[5] H Agrawal, R A DeMillo, and E H Spafford. An Execution Backtracking Approach to Program Debugging. *IEEE Software*, pages 21–26, 1991.

[6] H Agrawal and J R Horgan. Dynamic program slicing. In *Proceedings of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation*, pages 246–256. ACM, 1990.

[7] H Agrawal, J R Horgan, E W Krauser, and S A London. Incremental regression testing. In *Conference on Software Maintenance*, pages 348–357. Computer Society Press of the IEEE, 1993.

[8] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1985.

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

[10] Z.M. Ariola, M. Felleisen, J. Maraist, M. Odersky, and P. Wadler. A Call-By-Need Lambda Calculus. In P. Lee, editor, *Conference Record of the Twenty-Second Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 233–246. ACM, 1995.

[11] L. Augustsson. A compiler for lazy ML. In *Symposium on LISP and Functional Programming*, pages 218–227. ACM, 1984.

[12] H. Barendregt. *The Lambda Calculus: Its Syntax and Semantics*, volume 103 of *Studies in Logic and the Foundations of Mathematics*. Elsevier, revised edition, 1984.

[13] Lars Birkedal, Nick Rothwell, Mads Tofte, and David N. Turner. The ML Kit (Version 1). Technical Report 93/14, Department of Computer Science, University of Copenhagen, Universitetsparken 1, DK-2100 Copenhagen, 1993.

[14] S. Biswas. A demand-driven set-based analysis. In *Principles of Programming Languages*. ACM, 1997.

[15] G. Boudol. Computational semantics of term rewriting systems. In M. Nivat and J. C. Reynolds, editors, *Algebraic Methods in Semantics*. Cambridge University Press, 1985.

[16] R.S. Boyer and J.S. Moore. *A Computational Logic*. Academic Press, 1979.

[17] T. Chuang and B. Goldberg. Backward Analysis for Higher–Order Functions Using Inverse Images. Technical Report 620, Department of Computer Science, New York University, 1991.

[18] J. Ferrante, K. J. Ottenstein, and J. D. Warren. The program dependance graph and its use in optimization. *ACM Transactions on Programming Language and Systems*, 9(3):319–349, 1987.

[19] J. Field. On laziness and optimality in lambda interpreters: Tools for specification and analysis. In *Conference Record of the Seventeenth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 1–15. ACM, 1990.

[20] J. Field. A simple rewriting semantics for realistic imperative programs and its applications to program analysis. In *Proceedings of the ACM SIGPLAN Workshop on*

Partial Evaluation and Semantics-Based Program Manipulation, pages 98–107, 1992. Published as Yale University Technical Report YALEU/DCS/RR-909.

[21] J. Field, G. Ramalingam, and F. Tip. Parametric program slicing. In *Conference Record of the 22nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 379–392. ACM, 1995.

[22] J. Field and F. Tip. Dynamic dependence in term rewriting systems and its application to program slicing. Research report, IBM T.J. Watson Research Center, 1995. To appear.

[23] P. Fradet. Collecting More Garbage. In *ACM Conference on LISP and Functional Programming*, pages 24–33, 1994.

[24] M. Gandhe, G. Venkatesh, and A. Sanyal. Labeled λ-Calculus and a Generalised Notion of Strictness. In *Asian Computing Science Conference*, Lecture Notes in Computer Science. Springer-Verlag, 1995.

[25] C. A. Gunter. *Semantics of Programming Languages: Structures and Techniques*. Foundations of Computing. The MIT Press, 1992.

[26] C. Hanna and R. Levin. The Vesta Language for configuration management. Technical Report 107, Digital Equipment Corporation, Systems Research Center, 1993.

[27] J. Hannan and D. Miller. From operational semantics to abstract machines. *Mathematical Structures in Computer Science*, 2(4):415–459, 1992.

[28] N. Heintze. *Set Based Program Analysis*. PhD thesis, School of Computer Science, Carnegie Mellon University, 1992.

[29] N. Heintze. Set Based Analysis of Arithmetic. Technical Report CMU-CS-93-221, School of Computer Science, Carnegie Mellon University, 1993.

[30] N. Heintze. Set Based Analysis of ML Programs. In *Lisp and Functional Programming*, pages 306–317. ACM, 1994.

[31] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. In *Proceedings of the ACM SIGPLAN '88 Conference on Programming Language Design and Implementation*, pages 35–46. ACM, 1988.

[32] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. *ACM Transactions on Programming Language and Systems*, 12(1):26–90, 1990.

[33] P. Hudak and J. Young. Collecting interpretations of expressions. *ACM Transactions on Programming Languages and Systems*, 13:269–290, 1991.

[34] J. Hughes. Lazy Memo–functions. In J. Jouannaud, editor, *Proceedings of the Conference on Functional Programming and Computer Architecture*, pages 129–146. Springer-Verlag LNCS, 1985.

[35] J. Hughes. Compile–time Analysis of Functional Programs. In D. Turner, editor, *Research Topics in Functional Programming*, pages 117–153. Addison–Wesley, 1990.

[36] N. Jones and S. Muchnik. Flow analysis and optimization of LISP-like structures. In *Principles of Programming Languages*, pages 244–256. ACM, 1979.

[37] G. Kahn. Natural semantics. In *Proceedings of the Symposium on Theoretical Aspects of Computer Science*, pages 22–39. Springer-Verlag, 1987.

[38] J. Klop. Term rewriting systems. In S. Abramsky, D. M. Gabbay, and T. S. E. Maibaum, editors, *Handbook of Logic in Theoretical Computer Science*, pages 1–116. Oxford University Press, 1992.

[39] D.E. Knuth and P.B. Bendix. Simple Word Prolems in Universal Algebra. In J. Leech, editor, *Computational Problems in Abstract Algebra*, pages 263–297. Pergamon Press, 1970.

[40] B. Korel and J. Laski. Dynamic program slicing. *Information Processing Letters*, 29:155–163, 1988.

[41] P. Landin. The mechanical evaluation of expressions. *Comput. J.*, 6:308–320, 1964.

[42] J. Launchbury. A natural semantics for lazy evaluation. In S. L. Graham, editor, *Conference Record of the Twentieth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 144–154. ACM, 1993.

[43] T. Lengauer and R. Tarjan. A fast algorithm for finding dominators in a flowgraph. *ACM Transactions on Programming Languages and Systems*, 1(1):121–141, 1979.

[44] H.K.N Leung and H.K. Reghbati. Comments on program slicing. *IEEE Transactions on Software Engineering*, SE-13(12):1370–1371, 1987.

[45] R. Levin and P. McJones. The Vesta approach to precise configuration of large software systems. Technical Report 105, Digital Equipment Corporation, Systems Research Center, 1993.

[46] R. Milner and M. Tofte. Co–induction in relational semantics. *Theoretical Computer Science*, 87:209–220, 1991.

[47] R. Milner and M. Tofte. *Commentary on Standard ML*. The MIT Press, 1991.

[48] R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML*. The MIT Press, 1990.

[49] G. Morrisett, M. Felleisen, and R. Harper. Abstract models of memory management. In *Proceedings of the 7th International Conference on Functional Languages and Program Architecture*, pages 66–77, 1995.

[50] A. Mycroft. The Theory and Practice of Transforming Call-by-need into Call-by-value. In *International Symposium on Programming*. Springer-Verlag LNCS, 1980.

[51] L. C. Paulson. *ML for the Working Programmer*. Cambridge University Press, 1991.

[52] S. L. Peyton Jones. *The Implementation of Functional Programming Languages*. Prentice-Hall, 1987.

[53] G. D. Plotkin. Call-by-name, call-by-value and the lambda calculus. *Theoretical Computer Science*, 1:125–159, 1975.

[54] G. D. Plotkin. A structural approach to operational semantics. Technical Report FN-19, Computer Science Department, Aarhus University, Denmark, 1981.

[55] T. Reps and W. Yang. The semantics of program slicing. Technical Report 777, University of Wisconsin, 1988.

[56] J. A. Robinson. A machine oriented logic based on the resolution principle. *Journal of the ACM*, 12:23–41, 1965.

[57] N. Röjemo and C. Runciman. Lag, drag, void and use – heap profiling and space-efficient compilation revisited. In *Proceedings of the 1996 ACM SIGPLAN International Conference on Functional Programming*, pages 34–41, 1996.

[58] D. Schmidt. *Denotational Semantics: A Methodology for Language Development.* Allyn and Bacon, 1986.

[59] R.C. Sekar, P. Mishra, and I.V. Ramakrishnan. On the Power and Limitation of Strictness Analysis based on Abstract Interpretation. In *Conference Record of the 18th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 37–48. ACM, 1991.

[60] O. Shivers. Control flow analysis in scheme. In *Conference on Programming Language Design and Implementation.* ACM, 1988.

[61] O. Shivers. *Control Flow Analysis of Higher-Order Languages.* PhD thesis, School of Computer Science, Carnegie Mellon University, 1991.

[62] F. Tip. A survey of program slicing techniques. Research Report CS-R9438, Centrum voor Wiskunde en Informatica (CWI), 1994.

[63] G. A. Venkatesh. The semantic approach to program slicing. In B. Ryder, editor, *Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation*, pages 107–119. ACM, 1991.

[64] G. A. Venkatesh. Experimental results from dynamic slicing of C programs. *ACM Transactions on Programming Languages and Systems*, 17(2):197–216, 1995.

[65] M. Weiser. *Program Slices: Formal, psychological, and practical investigations of an automatic program abstraction method.* PhD thesis, University of Michigan, Ann Arbor, MI, 1979.

[66] M. Weiser. Programmers use slices when debugging. *Communications of the ACM*, 25:446–452, 1982.

[67] M. Weiser. Program slicing. *IEEE Transactions on Software Engineering*, SE-10(4):352–357, 1984.