Vrije Universiteit Brussel

Faculteit Wetenschappen
Vakgroep Computerwetenschappen
Software Languages Lab

Automatic Parallelization of Scheme Programs using Static Analysis

Proefschrift ingediend met het oog op het behalen
van de graad van Master in de Ingenieurswetenschappen

Jens Nicolay

Promotor: Prof. Dr. Theo D'Hondt
Begeleider: Dr. Coen De Roover

Academiejaar 2009 – 2010

Abstract

Automatic parallelization is an attractive option in light of the current proliferation of multicore hardware. This dissertation presents a brute-force technique for the automatic parallelization of sequential, higher-order Scheme programs containing side-effects. We will attempt to parallelize binding expressions inside nested `let` expressions by introducing a parallel `let` variant `let||`. To determine whether this is safe, we use dependency analysis. To facilitate the transformation to a parallel program, we can construct and transform a dependency graph.

If we want to determine dependencies between expressions in programs, we must analyze that program. Program analysis can be realized using abstract interpretation, because it provides a framework that allows us to collect information about a program without actually running it. The results produced by abstract interpretation are necessarily approximations because we are dealing with undecidable problems. Nevertheless our analyses are conservative and the results reflect all the possibilities that can arise at runtime.

In order to approximate dependencies that may arise when expressions invoke procedures in a higher-order program, we use the interprocedural dependence analysis by Might and Prabhu [19]to approximate dependencies that may arise when expressions invoke procedures in a higher-order program. The result of that analysis is an abstract resource dependency graph relating addresses and invocations. This information, together with the dependency information that is lexically apparent from the source code, allows us to determine whether it is safe to parallelize expressions in a program.

Our approach is brute-force in the sense that static analysis answers the safety question concerning parallelization, but we never investigate whether it is useful. We parallelize at every opportunity. To validate this approach and to see how it performs in practice, we ran several automatically parallelized benchmarks on parallel hardware. The evaluator we developed and use is Streme. It features parallelization primitives with intentional semantics. Our experimental results show that some programs with high levels of recursion indeed get a performance boost, while others do not or receive a performance penalty. The most promising future work comprises adding heuristics to guide the process of automatic parallelization.

To my unborn `(let ((f (lambda (x h) (if (zero? x) (h x) (lambda (f) (assoc f '((0 . son) (3 . daughter)))))))) (cdr (f 0 (f 3 #f))))`[1]

[1] I promised to keep this to myself. Reader discretion advised.

Acknowledgments

First of all, I would like to thank Prof. Theo D'Hondt for promoting this dissertation. Although I signed up for a thesis proposal with a somewhat different context than the one presented here, I still had the liberty to pursue my own interests. Which I did of course.

My gratitude also goes towards Prof. Matthew Might of the University of Utah for showing interest in my work, which is in large part based on his research. Prof. Might always found the time to answer my questions and provided me with suggestions that made the difference between getting it done or still being stuck.

Various people at the Software Lab helped me out. Charlotte Herzeel helped voicing my interest in using static analysis to attack parallelization. Stefan Marr was kind enough to set up an account for me on the "big machine". However, I am most indebted to my supervisor Coen De Roover for keeping me on track (well, for trying anyway). To put it simply: I could not have done it without his guidance and words of advice. Coen always encouraged me to do the necessary, rather than to wander off and explore every interesting idea I stumbled upon.

Last but not least, I want to thank my girlfriend Barbara for her continuing and loving support. I hope I can ever make up for the sacrifices she had to make while I was writing this dissertation.

Contents

List of Figures

Chapter 1

Introduction

1.1 Thesis

Parallelization of Scheme programs using static analysis is feasible and allows certain sequential programs to run faster on multicore systems.

1.2 Motivation

In 1965 Moore observed that in the previous years the number of components on a chip doubled about every 12 months. He also predicted that this trend would continue for at least ten years to come. In 1975 he revised his prediction and stated that complexity would double every 24 months [4]. Because his second prediction is still accurate even today, it has henceforth become known as Moore's Law. Chip manufacturers use it as a benchmark for their products and use the values it predicts as a roadmap for future efforts, and as such Moore's Law has become a kind of self-fulfilling prophecy.

Moore's Law has since been generalized to stating that hardware performance doubles about every 24 months as well. Proebsting's Law, on the other hand, makes the less optimistic observation that compilers double performance only every 18 *years* [24]. This means that the speedup over time gained by smarter compilers is negligible in comparison to the speedups caused by improved underlying hardware. It also means that there is still ample room for improvement as far as compiler optimizations are concerned. We believe that the recent trend of multicore desktop systems provides an opportunity for compilers to regain some of that lost ground.

In order to keep up with Moore's predictions, manufacturers of consumer-hardware chips have relentlessly been increasing CPU clock speeds. They did so well into the 21st century, until it became impractical to keep this up any further because of chip designs pushing against physical size and heat limits. Therefore, commercial manufacturers recently have turned to steadily increasing the number of processing cores inside a CPU to increase overall performance instead.

This kind of parallelism is not new. The first multiprocessor architectures date back to the 1960s. What *is* new, however, is the realization that parallelism is essential for continued progress in high-performance computing [8].

1.3 Problem

Increasing the clock speed virtually guarantees the speedup of the majority of sequential applications and programs. Increasing the number of cores to increase performance however usually comes at a price: programs in question must take advantage of the fact that they will run on parallel hardware.

We are now faced with two questions:

1. How should we go about writing new programs, knowing that they will be executed on parallel hardware?

2. What should we do with existing programs, of which the vast majority are sequential, so that they can benefit from running on multiple cores?

Starting with the first question, it may seem attractive to augment a programming language to allow for programmer directed or *manual parallelism*. This however leads us to a series of new questions [11]. In this dissertation we use Scheme as our programming language. Scheme is a small, elegant but yet expressive language. If people choose Scheme for one or more of these reasons, why should we lower the abstraction level of Scheme and complicate its simple semantics for the sake of parallelism? What are the specific annotations for parallelism we will add, and which aspects of parallelism will they capture? Can an average programmer uncover more parallelism and organize the computation more effectively than a compiler? Another argument against manual parallelization is the firm consensus that has grown on the fact that, especially in traditional imperative object-oriented languages, manual parallelization is tedious and error-prone [16].

Looking at the second question: who will rewrite all the existing sequential programs, if parallelization is to be done manually? There exists an immense body of sequential code, written over several decades in languages that do not necessarily support high-level parallelism.

Answering our two initial questions with “manual parallelization” therefore leads us to identify a series of problems from which we can conclude that manual parallelization it is not necessarily the easy option, and it is certainly not the only option.

1.4 Solution

Another approach would be to use *automatic parallelization*, where an optimizing compiler is the answer for effective programming of a parallel computer. This is the option this dissertation explores. Scheme as a language is an ideal candidate for this non-trivial undertaking because of its simplicity. We avoid cluttering up the Scheme itself with parallelization directives, retaining its simple semantics. We describe a method of transforming a sequential Scheme program into a parallel program while preserving its original semantics.

Automatic parallelization is difficult. Correctly applied manual parallelization often gives better results in specific cases than automatic parallelization does. The goal of this dissertation is not to prove otherwise, but to investigate how far we can get by using brute-force automatic parallelization, backed by static analysis.

While some of the results in this dissertation are still rudimentary and leave much room for improvement, we nevertheless hope that they represent a base on which further experiments can be conducted.

1.5 Overview

This dissertation describes an approach for the brute-force parallelization of sequential Scheme programs by using static analysis. More specifically, we look for nested `let` expressions and see if the expressions they bind can be evaluated in parallel (§2.2). If so, we can introduce a parallel `let` variant `let||`.

In order to guarantee the preservation of the sequential semantics of the input program, we must decide whether it is safe to parallelize expressions (§2.3). This depends on whether the expressions are dependent on each other or not. Therefore, we perform dependence analysis on all expressions appearing inside a structure of nested `let`s (§2.4).

The result can be represented by a dependency graph that expresses dependencies between different expressions (§2.5). Simple transformations that move the focus from dependency to evaluation order allow us to uncover and exploit opportunities for parallelization (§2.6). Our approach will be brute-force in the sense that we parallelize at every opportunity without asking if it is beneficial to introduce parallelism (§2.7).

Some types of dependence are lexically apparent and hence quite trivial in a reasonably simple language as Scheme, while others are not. Especially when procedure invocations in a higher-order setting are involved, interprocedural dependencies that are not always evident may arise and must be tracked.

The interprocedural dependence analysis we will use is based on stack reachability (§5.4.2). The underlying principle is that a procedure that calls another procedure is also dependent on the resources of its callee (§5.4.1). For the analysis to work, it needs to examine the call stack at the time when resources are read and written during evaluation.

If we want to obtain necessary but non-trivial information like possible runtime stack configurations from a program, it is no option to simply run that program (it may not terminate or depend on user input) or to try and predict information from the source code alone (the problems in general are undecidable). We can solve this problem by using the theory and techniques of abstract interpretation (§4.2). Hence it is abstract interpretation that powers the interprocedural dependence analysis.

If we want to parallelize expressions occurring inside nested `let` expressions, it is desirable to have as many of them as possible in the input program. To achieve this we employ a normal form, called ANF, that introduces many nested `let`s when converting to it (Chapter 3). ANF also simplifies abstract interpretation and therefore is used as the input language for the interprocedural dependence analysis as well.

To validate the approach and to test how brute-force automatic parallelization backed by static analysis performs, we describe several experiments and their outcome (Chapter 7). We use Streme as a Scheme evaluator to carry out these experiments. It implements low-level parallel constructs with intentional semantics, putting the evaluator instead of the programmer in charge of how much paral-

lelism a program will exhibit at runtime (§6.2). Parallelism in Streme is achieved using a work-stealing strategy (§6.3). Streme is an implementation of each and every idea described in this dissertation.

1.6 Contributions

The main contribution of this dissertation is the description and validation of and end-to-end framework consisting of static analysis and program transformation tools to implement *safe* brute-force parallelization at compile-time of sequential, higher-order Scheme programs in the presence of assignment through `set!` and side-effecting procedures (primitive or otherwise).

- We demonstrate that it is feasible to automatically introduce parallelism in sequential higher-order programs, with static analysis guaranteeing the preservation of the original semantics.
- We describe a detailed approach of how to analyze different types of dependencies inside a structure of nested `let` expressions, how to extract binding order constraints from those dependencies, and how to use these ordering constraints to generate parallel code.
- We implemented the interprocedural dependence analysis developed by Might and Prabhu [19] with minor modifications and extensions. Furthermore, we incorporated several techniques that optimize the analysis for speed without losing too much precision to become unuseful in practice.
- We developed a Scheme interpreter that is an implementation of all ideas and techniques presented in this dissertation.
  - The frontend is composed of a modular pipeline of analysis and transformation tools that can be modified and extended at will.
  - The backend consists of an evaluator that has parallelization primitives with intentional semantics, needed for our brute-force method of parallelization.
- We tested our dependency analysis, including the interprocedural analysis of Might and Prabhu, on several benchmarks using our Scheme interpreter, to validate our parallelization approach, check the validity of the analysis involved, and to verify its performance on real programs.

Chapter 2

Parallelization of Scheme Programs

2.1 Introduction

In this chapter we present our approach for parallelizing Scheme programs. The basic idea is simple: we look for nested `let` expressions and verify if it is possible to evaluate the expressions inside their bindings in parallel. We need to ensure that parallel evaluation of the binding expressions cannot alter the semantics of the program. To verify this we need to check for dependencies. Some dependencies are immediately apparent from the source code. Others require a more thorough analysis of the program.

2.2 Parallelization of nested `let` expressions

The approach we take to parallelize Scheme programs will consist of rewriting nested `let` expressions that may appear in programs. A `let` expression contains a list of zero or more *bindings* and a *body*. Bindings are of the form (u, e) where u is a variable and e the *binding expression*. When the `let` body is evaluated, the variables will be visible and bound to the value of their binding expressions.

Nested let expressions are a sequence of `let` expressions with one binding, chained through their bodies. The chain ends with the *innermost* `let` that does not have a `let` expression with one binding as its body. An example of a structure of nested `let`s with 3 bindings is

```
(let ((a e1))
  (let ((b e2))
    (let ((c e3))
      ebody))
```

In Scheme, nested `let`s can be conveniently expressed using `let*`. The above expression is equivalent with

```
(let* ((a e1)
       (b e2)
       (c e3))
  ebody))
```

Assume that we have a `let||` form that has identical semantics as an ordinary `let` except that it may evaluate its binding expressions in parallel. Basically, we want to know whether it is safe to rewrite

```
(let ((a e1))
  (let ((b e2))
    ebody))
```

as

```
(let|| ((a e1)
        (b e2))
  ebody)
```

If, after analysis, we have determined that it is possible to introduce `let||`, we have effectively parallelized (part of) our program.

More binding expressions in a nested `let` increase the opportunities for parallelization. Therefore, and also to allow for more uniform treatment, it is in our advantage if we bind the body expression of the innermost `let` to a variable, and use a reference to that variable to serve as the return value for the entire nested `let`. In examples we will use `body` as identifier for this purpose, which we assume to be unique. In practice this means that when confronted with expression

```
(let ((a e1))
  (let ((b e2))
    ebody))
```

we will actually rewrite this to

```
(let ((a e1))
  (let ((b e2))
    (let ((body ebody))
      body)))
```

with the innermost binding being the *body binding* and the result expression the *body reference*. To distinguish between body as a binding variable and as a reference, we will denote the reference as `=body`.

2.3 Dependence Analysis

We will use dependence analysis to decide whether it is *safe* to parallelize binding expressions appearing inside nested `let` expressions. The key principle in answering this question is *dependence*.

Definition 2.1. (Dependence) Expression e_2 is dependent on expression e_1 if e_1 must be evaluated before e_2 to guarantee the intended semantics of a program in which these expressions occur.

We can categorize dependencies into two classes: *control* dependencies and *data* dependencies.

Control dependencies

Expression e_2 is *control-dependent* on expression e_1 if the result of evaluating e_1 determines whether e_2 will be evaluated or not. For example, in the expression

```
(if (not (zero? x))
  (f (/ y x)))
```

the function `f` will only be called if the condition `(zero? x)` does not hold. Therefore there is a control dependency from `(f (/ y x))` upon `(not (zero? x))`. Control-dependent expressions cannot be reordered without changing the meaning of a program.

Data dependencies

Expression e_2 is *data-dependent* on expression e_1 when e_2 accesses or modifies a resource that is accessed or modified by e_1. In the context of this dissertation a resource will be a variable pointing to a fixed address containing a value that is modifiable. In the source code and in our dependency analysis we refer to a variable and its address by an identifier. We assume all identifiers are *unique* in the programs we analyze.

Let $R(e)$ return the set of variables read by expression e and $W(e)$ the set of addresses written by e. We can examine the following cases for expressions e_1 and e_2, where e_2 is sequentially executed after e_2:

- If $R(e_1) \cap R(e_2) \neq \emptyset$, then e_1 and e_2 share an address they read. This read/read data dependency is sometimes called an *input dependency* and is harmless in the sense that e_1 and e_2 can be reordered without changing the meaning of the program in which they appear. For example, in the following expression

  ```
  (let ((a 1))
    (display a)
    (set! x (* a a)))
  ```

  it does not matter whether the call to `display` comes before the assignment to `x` or after. In the remainder of this dissertation, when we state that two expressions are (data) dependent, we exclude this type of "dependency".

- If $W(e_1) \cap R(e_2) \neq \emptyset$, then e_2 reads an address that is written by e_1. This write/read is called a *true dependency* or a *flow dependency*. It means that e_2 must always be evaluated after e_1 and so the expressions cannot be reordered. In the expression

  ```
  (let ()
    (set! a 1)
    (set! b (+ a 1)))
  ```

  the assignment to `a` must always occur before the assignment to `b`, so the assignment is truly dependent on the binding.

- If $R(e_1) \cap W(e_2) \neq \emptyset$, then e_2 writes an address that is read by e_1. This read/write dependency is generally called an *anti-dependency* and is illustrated in the following expression, where the assignment to `a` is anti-dependent on the assignment to `b` .

  ```
  (let ((a 1))
    (set! b (+ a 1))
    (set! a 2))
  ```

  The assignments cannot be reordered without changing the meaning of the program.

- Finally, when $W(e_1) \cap W(e_2) \neq \emptyset$, then e_1 and e_2 share an address they write to. This write/write dependency is called an *output dependency* and again e_1 and e_2 cannot be reordered. In the expression

  ```
  (let ()
    (set! a 1)
    (display a)
    (set! a 2))
  ```

  the second assignment to `a` is output-dependent on the first assignment, and interchanging the assignments would alter the displayed value.

In short, expressions e_1 and e_2 cannot be reordered if

$$\{W(e_1) \cap \{R(e_2) \cup W(e_2)\}\} \cup \{W(e_2) \cap \{R(e_1) \cup W(e_1)\}\} \neq \emptyset$$

Implication for parallelization

So far we have identified different conditions which make it impossible to *reorder* expressions while preserving program semantics. The implication for *parallelization* follows naturally: two expressions that cannot be reordered can also not be executed in parallel. This is because parallelization possibly destroys the fixed sequential, non-overlapping evaluation order. Parallelization can preserve the sequential evaluation order of two expressions, but it can also reverse it. Besides reordering, parallelization can (and aims to) arrange for the evaluation of expressions to overlap, something which cannot occur during sequential evaluation.

Therefore, if a dependency exists between expressions e_1 and e_2, they cannot safely be evaluated in parallel. If e_2 does not depend on e_1, then evaluating these expressions in parallel cannot change the meaning of a program and hence it is safe to do so.

Dependence analysis with bindings

We already saw some examples of dependencies in Scheme programs in the previous section. The examples use a "classic" imperative setting, using assignment through `set!` to write to addresses. Scheme promotes a functional style of programming, including binding of variables (often instead of assigning) through `let` and its variations. We defined dependence in terms of expressions and a binding is technically not a Scheme expression. However, one can transform

```
(let ((a 1))
  (f a))
```

into

```
(let ((a undefined))
  (set! a 1)
  (f a))
```

where the let-binding serves more as an allocation of the address for variable `a` than a binding. By thinking of bindings as an allocation immediately followed by an assignment we can incorporate bindings in our dependence analysis without any additional difficulty.

2.4 Dependencies between nested `let` bindings

The precondition for successfully transforming programs by using a rewrite rule introducing `let||` is that there are no dependencies that prohibit us from doing so. To determine these dependencies we have to consider both control dependencies and data dependencies. In our approach however we will not handle control dependencies. In the expression

```
(let* ((a (if e_cond e_cons e_alt))
       (body e_body))
  body)
```

we disregard the control dependencies from e_{cons} and e_{alt} upon e_{cond} since we will not attempt to remove the if expression through any kind of transformation. This of course does not exclude the possibility of rewriting any nested `let`s that may appear *inside* if expressions.

Besides control dependencies we also have to look at data dependencies, and for this we assume that we have functions $R(e)$ and $W(e)$ at our disposal to determine the addresses that are read and written by binding expressions.

To avoid clutter and to simplify presentation, we often refer to bindings by their identifier only. We can do so because we assume variables have unique identifiers. Therefore, we can denote a dependency from binding (u_2, e_2) upon (u_1, e_1) as $u_1 \rightarrow u_2$. We also extend our read and write functions to accommodate for bindings. For binding (u, e) we have $R_B(u) = R(e)$ and $W_B(u) = W(e)$. Note that by convention function W_B only returns the addresses written by the binding expression, and never the binding variable itself.

Suppose (u_1, e_1) and (u_2, e_2) are two bindings inside a `let*`, and (u_2, e_2) is in the scope of (u_1, e_1). We can say that binding (u_2, e_2) is data-dependent on binding (u_1, e_1), or $u_1 \rightarrow u_2$, when one or both of the following conditions are met.

1. $u_1 \in \{R_B(u_2) \cup W_B(u_2)\}$, that is, e_2 reads and/or writes u_1 so we are dealing with either a flow or an output dependency (or both). We will refer to this type of dependency as a *scoping dependency*. For example, in the nested structure

   ```
   (let ((a 1))
     (let ((b (+ a 1)))
       (let ((body (* a b)))
   ```

```
      body)))
```

the expression bound to b reads variable a. Consequently, the binding for b is scope-dependent on the binding for a and b → a. Furthermore, the body binding is scope-dependent on both a and b, so $a \rightarrow body$ and $b \rightarrow body$. Scoping dependencies always target *internal* variables, bound in an enclosing let that is part of structure of nested lets being analyzed.

2. e_2 is dependent on e_1, and all types of data dependencies are possible. In the program fragment

```
(define z #f)

(let*
  ((x (if z 1 0))
   (y (set! z #t))
  ...)
```

$R_B(\mathtt{x}) = \{\mathtt{z}\}$ and $W_B(\mathtt{y}) = \{\mathtt{z}\}$ with variable z defined outside the nested let expressions. Hence, the binding for y is dependent on the binding for x or x → y. If we have

```
(let*
  ((z #f)
   (x (if z 1 0))
   (y (set! z #t))
  ...)
```

then the binding for y still is dependent on the binding for x, but now both these bindings are also scope-dependent on the binding for z, so we add dependencies x → z and y → z.

Scoping dependencies explain why the visibility of variables bound by let|| must necessary follow the same rules as that of a let and not let*. If e_2 accesses or modifies u_1, the scoping dependency that arises prohibits evaluating e_1 and e_2 in parallel. Hence it makes no sense to have let*-visibility for let|| even if we are using let|| to parallelize a series of nested lets.

Lambda expressions and procedure application complicate dependency matters further. We can make the following observations:

1. When binding a closure, we have to consider *only* the scoping dependencies appearing inside the lambda expression involved.

2. When binding the result of an invocation, we have to consider *all* dependencies appearing inside the lambda expression involved.

To illustrate the first observation, if we bind a closure we also have to look inside its lambda expression to determine scope dependencies. The fact that $R_B(\mathtt{incz}) = W_B(\mathtt{incz}) = \{\mathtt{z}\}$ and z → incz in

```
(let*
  ((z #f)
   (incz (lambda () (set! z 1) z)))
  ...)
```

may seem counter-intuitive when considering that the body of the lambda is not evaluated when `incz` is bound to a closure. But when reasoning in terms of lexical scoping and order, it is clear that `z` must be in scope and initialized before the closure is ever applied, so the bindings for `z` and `incz` cannot be reordered. When the dependency between two bindings is *not* a scoping dependency, as in

```
(lambda (z)
  (let*
    ((incz (lambda () (set! z 1) z))
     (squarez (lambda () (* z z))))
    ...))
```

then the reordering of the bindings for `incz` and `squarez` becomes permitted. Illustrating our second observation, we now bind the result of invoking `incz` and `squarez` as depicted below.

```
(lambda (z)
  (let*
    ((incz (lambda () (set! z 1) z))
     (squarez (lambda () (* z z)))
     (newz (incz))
     (zsquared (squarez)))
    ...))
```

We can still reorder the binding of the closures, but we cannot reorder the invocations because now we have to consider the dependencies between the bodies of the two lambda expressions as well. In this particular case we are confronted with a write/read dependency and as a consequence we have `newz` $\rightarrow$ `zsquared`. These type of dependencies that arise when procedures are invoked are called *interprocedural dependencies*. They are the most difficult dependencies to track, since they are not always lexically evident at the call sites (§).

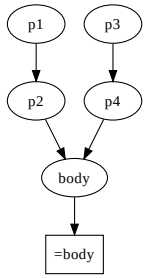
Besides control and data dependencies, there is one more form of dependency we have to take into account. In the expression

```
(let*
  ((z 42)
   (x (display z))
   (body (* z z))
  body)
```

we want the call to `display` to be finished before the body reference `=body` is returned, even though it does not contribute to the end result of the nested `let`. Still, to preserve the semantics of sequential evaluation we require `x` $\rightarrow$ `= body`. In general, if there is a binding (u, e) and there exists no other binding that is dependent on (u, e), we make the body reference dependent on (u, e). This guarantees that once the body reference is returned, all evaluations are terminated, as is always the case for a sequential evaluation of a `let`. In practice however, binding expressions in `let`s that do not (indirectly) contribute to the return value of the `let` but are evaluated for side-effects only (deliberate or not), are rare.

Figure 2.1 `fib` function and its binding dependency graph

```
(letrec
  ((fib (lambda (n)
          (let ((p0 (< n 2)))
            (if p0
                n
                (let ((p1 (- n 2)))
                  (let ((p2 (fib p1)))
                    (let ((p3 (- n 1)))
                      (let ((p4 (fib p3)))
                        (+ p2 p4)))))))))))
  (fib 40))
```



2.5 Binding dependency graph construction

We can use dependency information to construct a *binding dependency graph* that models the evaluation constraints imposed by the dependences between bindings appearing in our nested let expressions. This binding dependence graph is a directed, acyclic graph (DAG) where the nodes represent bindings, except for one unique node that represents the body reference

We will represent binding nodes using oval shapes. The body node is depicted in a rectangular shape and its identifier is prefixed with the symbol = to emphasize the fact that it is the return value. An edge from binding u_1 to u_2 in the dependency graph signifies that the binding of u_2 depends on u_1, or $u_1 \rightarrow u_2$ in short. An edge from binding u to the body node signifies that the body depends on u, either because of a scoping dependency or to guarantee that once the body reference is returned all evaluations are terminated. Because of this it is guaranteed that the body node will be the only sink node.

Example 2.2. (Binding dependency graph) In Figure 2.1 a double-recursive Fibonacci function is shown alongside its binding dependency graph. The bindings for `p1`, `p2`, `p3` and `p4` are readily identified in the source code, and `body` in our example is bound to `(+ p2 p4)`. The binding for `p0` does not feature in the graph since it is only involved in control dependencies. Variable `n` is not in the graph either, because it is not bound inside the nested `let`s we are analyzing. We have $R_B(\texttt{p2}) = \{\texttt{p1}\}$ so that `p1` $\rightarrow$ `p2`, $R_B(\texttt{p4}) = \{\texttt{p3}\}$ so that `p3` $\rightarrow$ `p4`, and $R_B(\texttt{body}) = \{\texttt{p2}, \texttt{p4}\}$ so that `p2` $\rightarrow$ `body` and `p4` $\rightarrow$ `body`.

From Example 2.2 it is clear that a binding dependency graph can have multiple source nodes (nodes for bindings `p1` and `p3`) but only one sink that contains the body reference `=body`. The example is also quite trivial, since the only dependencies that occur are bindings that read previously bound variables. The function `fib` does not read or write anything outside its lexical scope. When a procedure does access or modifies external resources, interprocedural dependencies complicate matters as Example 2.3 demonstrates.

Example 2.3. (Binding dependency graph with interprocedural dependency) In Figure 2.2 variable `z` plays the role of a common resource used in procedures `writez` and `readz`. The most interesting dependency is `p3` $\rightarrow$ `p4` since it expresses an interprocedural dependency resulting from the two

Figure 2.2 Nested lets with an interprocedural dependency and corresponding binding dependency graph

```
(let*
    ((z 0)
     (writez (lambda ()
               (set! z 123)))
     (readz (lambda () z))
     (p3 (writez))
     (p4 (readz)))
  (cons p3 p4))
```

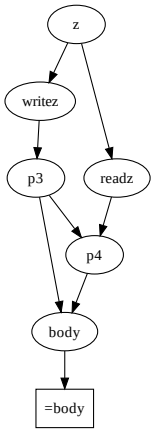
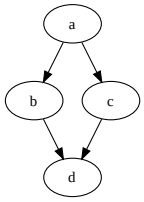


Figure 2.3 Parallelization opportunity in dependency graph

```
(let ((a e1))
  (let ((b e2)
    (let ((c e3))
      (let ((d e4))
        ...))))

                    ⇓

(let ((a e1))
    (let|| ((b e2)
            (c e3))
        (let ((d e4))
            ...)))
```



procedure applications. The calls `(writez)` and `(readz)` cannot be reordered because procedure `writez` writes to `z` and procedure `readz` reads from `z`.

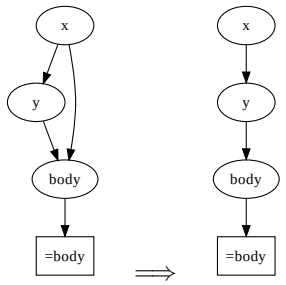
2.6 Identifying parallelism

If we have a binding dependency graph for a series of nested lets bindings, it becomes possible to identify opportunities for parallelism. The key idea is shown in Figure 2.3. Top-left we have a nested `let` with 4 bindings and a possible dependency graph for it on the right. There is a *branch* from binding `a` to bindings `b` and `c`. Since bindings `b` and `c` are independent, e_2 and e_3 can be evaluated in parallel when the evaluation of e_1 has finished. When the evaluation of *both* e_2 and e_3 is finished, e_4 can be evaluated. The code resulting from this graph is shown below the input program. The branch in the graph effectively has become a `let||` in the output program.

Dependence imposes ordering constraints, but the dependency graphs we have constructed so far are not always as straightforward to parallelize as the one in Figure 2.3, and therefore are not the most suitable representation for deciding an optimal binding order with respect to parallelization.

Figure 2.4 Pruning in dependency graph

```
(let*
    ((x 42)
     (y (* x x))
  (+ x y))
```



We can apply simple graph transformations like pruning and grouping to go from a binding *dependency* graph, explicitly representing all dependencies between bindings, towards a binding *order* graph that indicates the order in which binding can take place. The binding order graph can be viewed as a task dependency graph in which a task consists of binding one or more values of binding expressions to their variable.

Nodes are an ordered collection of bindings, except for the body node that is an ordered collection of bindings but can also contain the body reference `=body`. An edge $(u_1, \ldots, u_n) \rightarrow (u_{n+1}, \ldots, u_m)$ in a binding order graph signifies that bindings $u_1, \ldots, u_n$ must be evaluated before $u_{n+1}, \ldots, u_m$.

2.6.1 Pruning

Figure 2.4 shows a small program and its initial dependency graph. If we are interested in the order of binding, we can *prune* edge `x` $\rightarrow$ `z` because it is superfluous. If we have `x` $\rightarrow$ `y` $\rightarrow$ `body`, then the evaluation of `body` must always come after `y`. Pruning increases the opportunities for grouping, which we discuss next.

2.6.2 Grouping

If we want to make opportunities for parallelization more evident in our binding order graph, then we may also *group* operations. We will group operations from which we know that they must always be evaluated in a particular sequential order.

Figure 2.5 shows the `fib` function again. Grouping is accomplished by taking connected paths of nodes with exactly one entry point and one exit point. Grouping therefore is equivalent to basic block creation in traditional control flow diagrams. In the case of the Fibonacci function, the paths $p1 \rightarrow p2$ and $p3 \rightarrow p4$ can be grouped, as well as the `body` binding and its reference `=body`.

The fact that grouping makes parallelism more coarse-grained is not merely an additional benefit, but essential if we want to avoid unnecessary synchronization points during parallel evaluation, resulting in excessive overhead. Therefore we strive to make groups as large as possible.

To see why pruning must occur before grouping, consider the binding order graph after pruning and grouping in Figure 2.6. The program in this figure the same as in Figure 2.4, and it is the removal of edge $x \rightarrow z$ that allows us to group all bindings in one node. If a binding dependency graph can be reduced to a single node in this fashion, this effectively means that the nested `let`s are *not parallelizable*.

Figure 2.5 Grouping in dependency graph

```
(letrec
  ((fib (lambda (n)
          (let ((p0 (< n 2)))
            (if p0
                n
                (let ((p1 (- n 2)))
                  (let ((p2 (fib p1)))
                    (let ((p3 (- n 1)))
                      (let ((p4 (fib p3)))
                        (+ p2 p4)))))))))))
  (fib 40))
```

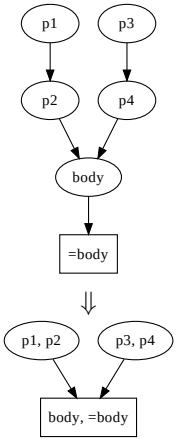


Figure 2.6 Graph after pruning and compacting

```
(let*
    ((x 42)
     (y (* x x))
  (+ x y))
```

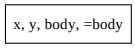
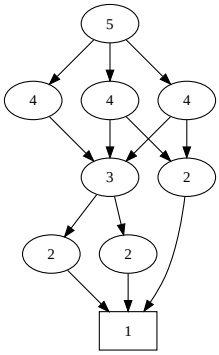


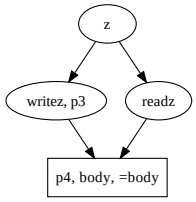
Figure 2.7 Topological sorting with node labels indicating node level



2.7 Generating code

Once we have exhaustively applied pruning and grouping, the resulting binding order graph is in a form that allows straightforward conversion into Scheme code. To determine the final binding order, we divide the graph in layers by applying a *topological sort*. In a topological sort, every node is assigned a rank or level, and all nodes of the same level form a *binding layer*. The level of a node is equal to the number of nodes in the *longest* path possible from that node to the sink node. The sink node trivially has level 1. Figure 2.7 shows an example of a topological sort. Note that we need an unique sink node to make topological sorting possible. Our approach with the unique body node that is made dependent on all other binding nodes that would otherwise be sink nodes to guarantee termination of binding expression evaluation, assures this.

Figure 2.8 Binding order graph and code generated from the graph




```
(let* ((z 0))
  (let ((writez undefined))
    (let|| ((p3 (begin
                  (set! writez (lambda ()
                                 (set! z 123)))
                  (writez)))
            (readz (lambda () z)))
      (let* ((p4 (readz))
             (body (cons p3 p4)))
        body))))
```

Binding nodes together with the topological sort order determines the global order in which binding can take place. These are the key insights.

1. All bindings inside a single binding node *must be evaluated sequentially* (because of the way we defined the grouping operation).

2. A group of bindings contained inside a single binding node *can be evaluated in parallel* with other groups of bindings inside the same binding layer.

Talking in terms of code generation, binding nodes with the same level will be translated into Scheme code using `let||`.

To illustrate this, consider the program from Example 2.3 again with its binding order graph, depicted in Figure 2.8. Topological ordering gives us the following binding layers:

- Level 3: bindings `z`
- Level 2: binding group (`writez`, `p3`), binding `readz`
- Level 1: binding group (`p4`, `body`) with body reference

Level 2 contains multiple nodes, and therefore binding group (`writez`, `p3`) and binding `readz` can be evaluated in parallel. Level 1 and level 3 only contain a single binding node, so there is nothing to evaluate in parallel, and the bindings inside these nodes are evaluated sequentially.

Because binding group (`writez`, `p3`) appears as a branch inside `let||`, but a binding only declares one variable, `p3` will be used as variable for the `let||` binding. The declaration of other the binding, `writez`, is lifted outside the `let||` to an immediately enclosing `let`.

2.8 Conclusion

In this chapter we outlined our approach for parallelizing sequential Scheme programs. We will attempt to parallelize binding expressions inside nested `let` expressions by introducing their parallel variant `let||`. To determine whether this is safe, we use dependency analysis. Any independent pair of expressions can be reordered, and hence can be parallelized. To facilitate the transformation from nested `let`s to a `let||`, we can construct a binding dependency graph with bindings as nodes and

edges indicating dependencies. We transform this graph into binding order graph by pruning edges and grouping nodes. Making ordering constraints explicit, the binding order graph is suitable structure for generating parallel code.

In this chapter we made several assumptions that we still need to address.

- We assumed that we have functions $R(e)$ and $W(e)$ at our disposal, returning the set of addresses read and written, respectively, by an expression. Chapter 5 shows how we can obtain these functions, even in the presence of side-effecting higher-order procedures.
- Our target for parallelization is a structure of nested `let` expressions. We presented the Fibonacci function several times as an example, written using nested `let`s. It is doubtful however that input programs will be presented to an evaluator (exclusively) in this style. We need a transformation that introduces nested lets and increases our targets for parallelization. Such a transformation is presented in Chapter 3.

Chapter 3

Administrative Normal Form

3.1 Introduction

Static analysis, program transformation and code generation is often performed on a transformed version of a program instead of the original source code, called *intermediary forms*, to facilitate the work that needs to be done. Two of the most widely known and used intermediary forms are continuation-passing style (CPS) [30] and administrative normal form (ANF) [10].

In this dissertation we will use ANF as intermediary form. We will define ANF and give some examples, and mention the fact that ANF helps in realizing our approach as explained in Chapter 2. Other advantages of ANF however, only come into play in later chapters so we cannot provide the full details here yet. Therefore, inevitably, this short chapter contains many forward references. As such, the reader may find it useful to consider this chapter as a reference (especially Figure 3.2) and consult back here while reading later chapters.

3.2 Definition

ANF conversion simplifies expressions in the sense that it breaks them down into simpler subexpressions until no further simplification is possible and the code effectively has become a normal form.

The principle mechanism of simplification is that every intermediate result is named using a `let`. For example, the expression

```
(/ (+ (sqrt (- (* b b) (* 4 a c)))) (* 2 a))
```

is broken down into

```
(let ((_p0 (* b b)))
  (let ((_p1 (* 4 a c)))
    (let ((_p2 (- _p0 _p1)))
      (let ((_p3 (sqrt _p2)))
        (let ((_p4 (+ _p3)))
          (let ((_p5 (* 2 a)))
            (/ _p4 _p5)))))))
```

Figure 3.1 Example of a recursive factorial function in ANF

```
(define (fact n)
  (let
    ((t1 (zero? n)))
   (if t1
     1
     (let
       ((t2 (- n 1)))
      (let
        ((t3 (fact t2)))
       (* n t3))))))
```

where variables with prefix `_p` are unique variables introduced during the conversion process. The result of ANF conversion is that every function call is either `let`-bound or is a tail call. All operands in function calls are trivial expressions.

Conversion also reaches inside conditional expressions, declarations mutations and procedure bodies. The expression

```
(if (f x) (g (* y1 y2)) (+ z1 z2))
```

becomes

```
(let ((_p0 (f x)))
  (if _p0
      (let ((_p1 (* y1 y2)))
        (g _p1))
      (+ z1 z2)))
```

in ANF. Note that expression `(+ z1 z2)` appears unaltered in the ANF version because it is a tail call.

Figure 3.1 shows an example of a recursive factorial program expressed in ANF.

Finally, the ANF language used in this dissertation is presented in Figure 3.2.

3.3 Motivation

We choose ANF over CPS or any other intermediary form because ANF has some nice properties that fit well into our approach.

1. It has the property of introducing many nested `let`s.
2. It will make subsequent interprocedural analysis considerably less complex.
3. ANF preserves the direct-style behavior of the original.

Because our target for parallelization are nested `let`s that occur inside a Scheme program, it is clear that the first point makes ANF a natural choice as intermediary form to base our automatic parallelization on.

The second point will become evident when we describe the actual interprocedural analysis needed to safely evaluate binding expressions occurring inside nested `let`s in parallel. The argument can

Figure 3.2 Administrative normal form with recursive functions, mutable variables and conditional.

$$
\begin{array}{rcl}
u \in \mathsf{Var} & = & \text{set of identifiers} \\
q \in \mathsf{Quo} & = & \text{set of literals} \\
p \in \mathsf{Prim} & = & \text{set of primitives} \\
lam \in \mathsf{Lam} & ::= & (\lambda(u_1 \ldots u_n)\ e_{body}) \\
proc \in \mathsf{Proc} & = & \mathsf{Lam} + \mathsf{Prim} \\
f, x \in \mathsf{Arg} & = & \mathsf{Var} + \mathsf{Quo} + \mathsf{Proc} \\
e \in \mathsf{Exp} & ::= & x \\
& | & (f\ x_1 \ldots x_n) \\
& | & (\texttt{let}\ ((u\ x))\ e_{body}) \\
& | & (\texttt{let}\ ((u\ (f\ x_1 \ldots x_n)))\ e_{body}) \\
& | & (\texttt{letrec}\ ((u\ lam))\ e_{body}) \\
& | & (\texttt{set!}\ u\ x) \\
& | & (\texttt{if}\ x_{cond}\ e_{cons}\ e_{alt})
\end{array}
$$

be made however without knowing any details of the analysis, except that it involves some form of interpretation (§4.3) of Scheme programs. Looking at procedure application as the prime example, ANF ensures that all operands are trivial or *atomic expressions* (Definition 5.11) that can be evaluated in finite time without side effects. This means that an interpreter does not need to "step into" the operands to evaluate them by recursively calling a full `evaluate` on them. Instead, an interpreter can use a simpler `atomic-evaluate` procedure and map it over the operands, simplifying the semantics as we will see in §5.4.6.

The third and last point has expresses the fact that we want to retain the original direct-style behavior of an input program. For our purposes, we can directly oppose direct-style to continuation-passing style. CPS programs are characterized by the absence of return flow. However, the basic principle that we will use for determining interprocedural dependencies depends on a control flow that can be captured by a traditional call stack (§5.4.1). Therefore it is important that ANF is a direct-style language.

3.4 Conclusion

Administrative normal form brings many advantages when used in our approach. Converting to ANF essentially consists of breaking down expressions into their subexpressions and naming those subexpressions by introducing `let` expressions. As such, converting to ANF increases the number of targets that we can attempt to parallelize with our approach described in Chapter 2. It also simplifies subsequent program analysis, and by preserving the direct-style behavior of programs it does not invalidate the analysis we present in Chapter 5.

Chapter 4

Static Analysis

4.1 Introduction

In order to implement our approach as described in Chapter 2, we need to be able to determine the addresses read and written by expressions in a program during evaluation. In the examples given in that chapter this was always quite trivial, but there are more difficult cases that we must solve.

- We want to know what values an expression may evaluate to (*value flow*) and which procedures can be applied at call sites (*control flow*). Consider for example expression `(f x)`. The actual procedure that is invoked depends on variable `f`. But the value of `f` may depend on a procedure invocation itself. Traditional first-order flow analysis techniques for imperative Algol-class languages do not work for Scheme because the operator in a function call can be any kind of expression that evaluates to a procedure [29].
- When confronted with a procedure invocation such as `(f x)`, we have to look at the resulting *interprocedural dependencies*. We want to know what addresses are read and written at a call site. Syntactically different occurrences of expression `(f x)` can yield different dependencies during invocation. One occurrence of such an expression in a program can access or modify different resources over time if it is evaluated more than once [19].

There are several options we can try to obtain answers to the above questions.

- We could simply run the program and keep track of relevant information, but we may find that it does not terminate, that it terminates with an error, or that the outcome is different for different user inputs.
- We could also look at the source code, but then we are up against the halting problem. We cannot decide whether the program will terminate or run forever, not even when given a specific input. We can not even decide whether it possesses any non-trivial property whatsoever. The concrete semantics of a program – the set of all possible behaviors of that program when executed for all possible input data – are in general not computable[6].

Clearly both undertakings are problematic. The rest of this chapter is devoted to the theory and techniques to find a way around these obstacles.

4.2 Abstract Interpretation

Abstract interpretation offers a solution to the problem of uncomputable run-time properties. Although the concrete semantics of a non-trivial program are not computable with finite resources, abstract interpretation nevertheless offers a framework for getting useful answers to non-trivial questions about programs. These answers will be necessarily be approximations due to the undecidability of the questions asked.

Abstract interpretation as a formal method for the analysis of programs was pioneered by Cousot and Cousot [5].

4.2.1 Abstractions and concretizations

Abstraction is the single most important tool of a programmer, and therefore every programming language essentially is a collection of abstraction mechanisms at the disposal of the programmer. Abstraction allows one to focus on one part of a problem while abstracting away other details.

Value abstractions are often encountered in computing [26]. For example, if $x \sqsubseteq y$ denotes that y is less precise than x, then $\langle 2 \rangle \sqsubseteq \langle 2, 5 \rangle \sqsubseteq \langle 0..9 \rangle \sqsubseteq \widehat{\texttt{int}}$ are all abstractions of the number 2. An abstract value "names" or represents one or more concrete values. We denote abstract values with angle brackets that delimit a set of concrete values, e.g. $\langle 2, 5 \rangle$, or as a name with a "hat", e.g. $\widehat{\texttt{int}}$.

The concretization function γ maps an abstract value to the set of concrete values it represents. For example, $\gamma(\langle 2, 5 \rangle) = \{2, 5\}$ and $\gamma(\widehat{\texttt{int}}) = \{\ldots, -1, 0, 1, \ldots\}$. Conversely, the abstraction function α maps a set of concrete values to an abstract value that offers the most precise *approximation*. If again$\langle 2 \rangle$, $\langle 2, 5 \rangle$, $\langle 0 \ldots 9 \rangle$, and $\widehat{\texttt{int}}$ are the available abstract values, then $\alpha(\{2\}) = \langle 2 \rangle$ and $\alpha(\{1, 2, 3, 42\}) = \widehat{\texttt{int}}$.

When dealing with approximations, the abstraction mapping α introduces *imprecision*. This is a consequence of the fact that we are trying to compress infinitely many concrete values into a finite set of abstract values. For example, we can have $\alpha(\{2, 5, 6\}) = \langle 0 \ldots 9 \rangle$ but $\gamma(\langle 0 \ldots 9 \rangle) = \{0, 1, \ldots, 9\}$, so that $(\gamma \circ \alpha)(\{2, 5, 6\}) \neq \{2, 5, 6\}$. Because $\{2, 5, 6\} \subseteq \{0, 1, \ldots, 9\}$ we say that α is *sound*. While $(\gamma \circ \alpha)$ is inexact, it is easy to verify that the inverse $(\alpha \circ \gamma)$ never loses precision.

More formally, if for a set of concrete values X, with $\mathcal{P}(X)$ denoting the power set of X, and its abstract counterpart $\hat{X}$ we have:

$$\forall c \in \mathcal{P}(X), a \in \hat{X} : c \subseteq \gamma(a) \iff \alpha(c) \sqsubseteq a$$

then the abstraction is sound. The above relation between X and $\hat{X}$ is called a *Galois connection*.

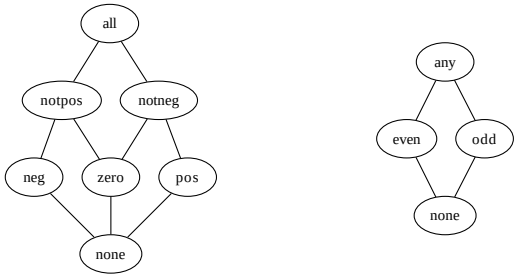
4.2.2 Lattices

Our example of abstracting the number 2 used a very simple ordered set of abstract values. We now give a mathematical characterization of the sets involved in abstract interpretation [27, 26].

In general, if $\sqsubseteq$ is a binary relation over a set X for which the following holds

- $\forall x \in X : x \sqsubseteq x$ (reflexivity)
- $\forall x, y \in X : x \sqsubseteq y \wedge y \sqsubseteq x \Rightarrow x = y$ (antisymmetry)

Figure 4.1 Two examples of complete lattices



(a) (b)

- $\forall x, y, z \in X : x \sqsubseteq y \wedge y \sqsubseteq z \Rightarrow x \sqsubseteq z$ (transitivity)

then $(X, \sqsubseteq)$ is called a *partial order*, and X is a *partially ordered set*.

If $Y \subseteq X$, then $x \in X$ is an *upper bound* for Y, written $Y \sqsubseteq x$ if $\forall y \in Y : y \sqsubseteq x$. Similarly, $x \in X$ is a *lower bound* for Y, written $x \sqsubseteq Y$, if $\forall y \in Y : x \sqsubseteq y$. A *least* upper bound, written $\sqcup Y$, is defined by:

$$Y \sqsubseteq \sqcup Y \wedge \forall x \in X : Y \sqsubseteq x \implies \sqcup Y \sqsubseteq x$$

Similarly a *greatest* lower bound, written $\sqcap Y$, is defined by:

$$\sqcap Y \sqsubseteq Y \wedge \forall x \in X : x \sqsubseteq Y \implies x \sqsubseteq \sqcap Y$$

In a partial order we have $x \sqsubseteq y$ or $y \sqsubseteq x$ if x and y are comparable, else x and y are incomparable. If we make the stronger assumption that any two elements of a partially ordered set have both a unique least upper bound (supremum) and a unique greatest lower bound (infimum), we obtain a *lattice*. If $(X, \sqsubseteq)$ is a lattice and every $Y \subseteq X$ has a least upper bound and a greatest lower bound, then X is a *complete lattice*. A complete lattice must have a unique largest element called "top", denoted $\top$ and defined as $\top = \sqcup X$, and a least elements called "bottom", denoted $\bot$ and defined as $\bot = \sqcap X$. It follows that for all $x \in X$ we have $\bot \sqsubseteq x$ and $x \sqsubseteq \top$.

Example 4.1. (Lattice) Figure 4.1 shows two examples of complete lattices [26]. Both are possible abstractions of $(\mathcal{P}(\mathbb{Z}), \subseteq)$, which is a complete lattice itself. In Figure 4.1 a the top element is `all` and the bottom element is `none`. The least upperbound of $\{$`notpos`, `notneg`$\}$ is `zero`. The least upper bound of $\{$`zero`, `notpos`, `notneg`$\}$ is `all`.

Abstractions are not limited to values but also extend to operations. Suppose we have a concrete operation f defined on concrete domain X. $\hat{f}$ is a sound abstraction of f if the following holds [7]:

$$\forall x \in X : f(x) \sqsubseteq \gamma(\hat{f}(\alpha(x)))$$

or, equivalently,

$$\forall \hat{x} \in \hat{X} : \alpha(f(\gamma(\hat{x}))) \sqsubseteq \hat{f}(\hat{x})$$

Example 4.2. (Sign abstraction) A widely used example is the abstraction of integers to their sign [5, 21]. The concrete values are the integers $\mathbb{Z}$, while the abstract values are in the power set of signs

$\hat{\mathbb{Z}} = \mathcal{P}(\{-, 0, +\})$. For every $z \in \mathbb{Z}$ we have

$$\alpha(z) = \begin{cases} \langle - \rangle & z < 0 \\ \langle 0 \rangle & z = 0 \\ \langle + \rangle & z > 0 \end{cases}$$

The addition operator $+ : \mathbb{Z} \times \mathbb{Z} \to \mathbb{Z}$ abstracts naturally to $\oplus : \hat{\mathbb{Z}} \times \hat{\mathbb{Z}} \to \hat{\mathbb{Z}}$ following the rule of signs, so that for example:

$$\begin{aligned} \langle 0 \rangle \oplus \langle 0, + \rangle &= \langle 0, + \rangle \\ \langle + \rangle \oplus \langle + \rangle &= \langle + \rangle \\ \langle + \rangle \oplus \langle - \rangle &= \langle -, 0, + \rangle \\ \langle +, - \rangle \oplus \langle 0 \rangle &= \langle -, + \rangle \end{aligned}$$

The abstract execution of $4 + (-4)$ yields $\langle -, 0, + \rangle$, while concretely evaluating $4 + (-4)$ and then abstracting would yield $\alpha(4+(-4)) = \langle 0 \rangle$, showing that abstract interpretation strictly over-approximated. Furthermore, $\{4 + (-4)\} = \{0\} \subseteq \{-, 0, +\} = \gamma(\langle + \rangle \oplus \langle - \rangle)$ supports the fact that $\oplus$ is a sound abstraction of $+$.

4.3 Static analysis via abstract interpretation

Historically, much effort has already been directed towards predicting the run-time behavior of programs without actually running those programs. The source code of a program, as opposed to a run of the program it expresses, is static and does not depend on external input. Hence it can be analyzed in finite time with finite resources, although we must settle for approximations when computing run-time properties if we want to ensure termination of our algorithms.

Abstract interpretation is a good match for static analysis, because it provides a theory to develop static analyses that are correct and provide meaningful results about the run-time behavior of programs.

In the context of this dissertation we want static analysis to *conservatively approximate* the addresses read and written when procedures are invoked to determine interprocedural dependencies. Conservative means erring on the safe side in order to remain faithful to the original semantics. If during evaluation a resource is accessed or modified while applying a procedure, the analysis must reflect this possibility, but the analysis may be imprecise in the sense that it overestimates the set of resources read or written at a specific call site.

4.3.1 Techniques

Static analysis via abstract interpretation in this dissertation is performed on an abstract domain that mimics the concrete domain. The general strategy is as follows:

- We define the concrete semantics for Scheme that allows us to compute $R(e)$ and $W(e)$.
- We define the abstract semantics that correlates with the concrete semantics.

- We construct an algorithm to finitely and efficiently compute approximations of the abstract semantics.

The concrete semantics will be modeled by an infinite-state machine that captures the concrete semantics of Scheme. Next we approximate this state machine with a finite-state machine to obtain an abstract interpreter that determines one or more properties of programs that it is capable of running.

The abstract interpretations used for value flow and dependence analyses we will encounter are of the *collecting* type. The goal is to derive information from source code that is both specific enough to permit certain optimizations like parallelization and general enough to be valid in every state in which the program may be executed [12].

The abstract interpreter continually transforms abstract machine states into possible successor states, generating a graph of all potentially reachable states. When a fixpoint is reached (i.e. all successor states are already in the graph), the interpreter crawls the graph of machine states (starting from the initial state) collecting the results we are interested in. We build such an abstract interpreter in §5.4.

In statically typed languages, the abstractions used often follows the structure of the types. But in the absence of static types, like in a dynamically typed language as Scheme, abstract domains are defined in terms of *program points* [22]. Depending on the analysis, a program point can be any combination of procedure application points, procedure entry and exit points, lambda expressions, etc. Usually program points are *labeled* to more easily denote and distinguish different program point instances throughout a program.

4.3.2 Performance of analysis

Abstract interpretation is a (carefully) designed exchange: in order to calculate some properties of a program in a reasonable amount of time (speed of the analysis), we are willing to make a trade-off in terms of lower precision and power to be able to do so. From this we can define the three axes across which the performance of an analysis can be ranked [21]:

- *Speed* refers to the running time of an analysis. Although speed is important for a real-world implementation, this will not be our major concern in this dissertation. Of course, every analysis must terminate.

- *Precision* refers to how tight an analysis constrains a set of answers to the set of actually possible answers. Both “yes” and “no” are more precise as “maybe”, and $\langle\lambda_{42}\rangle$ is more precise than “the set of all lambdas in a program”.

- *Power* refers to the class of questions which an analysis can answer. For example: an analysis that can answer must-alias questions is more powerful than a may-alias analysis.

4.4 Conclusion

If we want to determine dependencies between expressions in all but the most trivial of programs, we must be able to analyze that program in a reasonable (finite) amount of time. Program analysis can be

realized using abstract interpretation to ensure termination of our algorithms. Abstract interpretation provides a framework that allows us to collect information about a program before actually running it, on which we can base the results of our analysis. These results are approximations but nevertheless reflect all the possibilities that can arise during program evaluation.

Armed with the technique of abstract interpretation, we can now give a detailed description of the actual analysis that allows us to detect the more difficult types of dependencies present in higher-order Scheme programs. This is the task we set ourselves in Chapter 5.

Chapter 5

Interprocedural Dependence Analysis

5.1 Introduction

In order to perform dependency analysis we need to know which addresses are read and written by expressions that use values through aliasing or invoke procedures. This requires the conservative approximation of the value and control flow of a program to determine which procedures are invoked at specific call sites at specific points during evaluation. Additionally, at each of these call sites we want to conservatively bound the addresses read and written by the procedure that is invoked.

Might and Prabhu developed an abstract interpretation of ANF that is capable of computing the required information. The result is context-sensitive interprocedural dependence analysis (IPDA) for higher-order programs [19].

5.2 Abstract resource dependence graphs

The most important output of IPDA is the *abstract resource dependence graph*[1]. The abstract resource dependence graph relates abstract resources (representing a set of concrete resources) with abstract procedure invocations. Abstract resources in the context of this dissertation will be addresses of variables. We will depict addresses in square nodes. An abstract procedure invocation is a procedure plus an abstract calling context. The context serves to discriminate between different invocations of the same procedure.

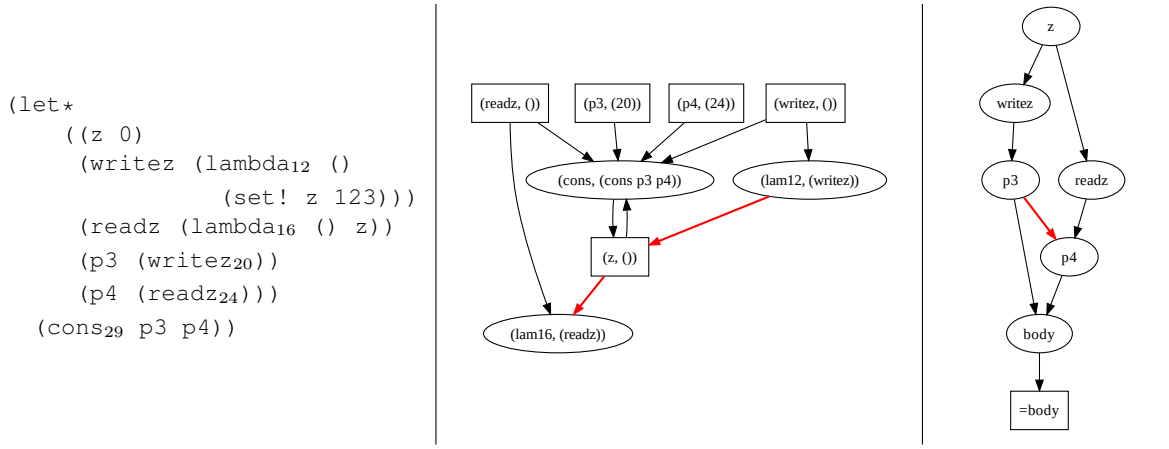
An edge from an abstract resource to an abstract invocation means that, during the execution of that procedure in the given context, the resource may be read. An edge from an abstract invocation to an abstract resource indicates the possibility that, during the execution of that procedure in the given context, the resource may be written.

Constructing the graph in this fashion allows us to uncover four types of interprocedural dependencies between two abstract invocations. These are exactly the types of dependencies we encountered in §2.3.

- read/read dependency: two invocations can be reached from the same abstract resource.

[1] Whenever we mention "dependence graph" in this chapter, we mean "abstract resource dependence graph" (not to be confused with the dependency graphs from Chapter 2).

Figure 5.1 Nested lets with corresponding abstract resource dependency graph (middle) and binding dependency graph (right)



- read/write dependency: there is a path from the second invocation to the first invocation.
- write/read dependency: there is a path from the first invocation to the second invocation.
- write/write dependency: two invocations reach the same abstract resource.

Example 5.1. (Abstract resource dependence graph) Figure 5.1 shows the same program from Example 2.3 on page 12, together with its abstract resource graph and binding dependency graph. The edge $(\lambda_{12}, (\texttt{writez})) \rightarrow (\texttt{z}, ())$ in the abstract resource graph indicates that procedure λ_{12} writes variable `z` when invoked at call site 20. The edge $(\texttt{z}, ()) \rightarrow (\lambda_{16}, (\texttt{readz}))$ indicates that procedure λ_{16} reads variable `z` when invoked at call site 24. Because there is a path from $(\lambda_{12}, (\texttt{writez}))$ to $(\lambda_{16}, (\texttt{readz}))$ via $(\texttt{z}, ())$, this means that there effectively is an interprocedural write/read dependency between the invocations `(writez)` and `(readz)`. This results in the dependency `p2` $\rightarrow$ `p3` in the binding dependency graph.

Similarly to what we observed in §2.3, read/read dependencies can be considered harmless in the setting because no synchronization is needed between two procedure invocations that exhibit this type of dependency. The other three dependences between invocations are not safe to parallelize without synchronization: the order of the invocations must be preserved to guarantee that the semantics of the program are unaltered when the invocations are allowed to run in parallel.

5.3 Context sensitivity

Different resources can be read or written when the same procedure is invoked at different application sites. IPDA can be made context-sensitive, allowing us to separately track dependences for the same procedure when invoked at different call sites.

The abstract resource graph in Figure 5.1 already attaches a calling context to each procedure. In that example however we do not really benefit from this information because there is no ambiguity even without the additional calling contexts. Example 5.2 shows how context-sensitivity can improve the analysis.

Figure 5.2 Context-sensitive dependence analysis.

```
(define a #f)
(define b #f)

(define (write-a) (set! a #t))
(define (write-b) (set! b #t))

(define (unthunk f) (f))

(unthunk write-a)
(unthunk write-b)
```

Example 5.2. (Context-sensitive dependence analysis) In the example in Figure 5.2, taken from [19], the first call to `unthunk` is write-dependent on `a` and the second call is write-dependent on `b`. It is only by including context information that the dependence analysis is able to realize that the two calls to `unthunk` are not dependent on each other. In a context-insensitive setting both calls of `unthunk` would be seen as having a write dependency on both `a` and `b`.

5.4 Computing the analysis

5.4.1 Harrison's Principle

Might and Prabhu's interprocedural analysis was inspired by Harrison's dependence principle, in which *procedure strings* are the central semantic concept. Procedure strings capture the control flow of procedure applications by chaining the activation and deactivation of procedures throughout the execution of a program. Putting this into notation, every application of a procedure is registered as a downward movement using superscript d. Control moves upwards when a procedure returns, indicated by a superscript u. Suppose a program contains three procedures λ_α, λ_β, and λ_γ. A possible procedure string after six interprocedural movements could be $p_6 = \alpha^d\beta^d\beta^u\alpha^d\gamma^d\alpha^d$, meaning that there have been three applications of λ_α, and one of λ_β and λ_γ each. λ_β is no longer active in p_6. We can observe facts like these more clearly by canceling all matching activations and deactivations from a procedure string. In our case the pair $\beta^d\beta^u$ cancels out, yielding the balanced procedure string $Net(p_6) = \alpha^d\alpha^d\gamma^d\alpha^d$.

Using subtraction, $p_y - p_x$ denotes the activity that occurred between two states of a program where p_x and p_y were captured. If p_x corresponds with the birth date of a resource, and p_y with the use (reading or writing) of a resource, then $Net(p_y - p_x)$ gives all the procedures that are live on the stack and hence have a dependency on that resource. This last observation forms the basis of Harrison's interprocedural dependence analysis, which leads us to the definition of the following principle.

Definition 5.3. (Harrison's Dependence Principle) Assuming that a program obeys the traditional call stack behavior, when a resource is read or written, all of the procedures which have frames live on the stack have a dependence on that resource.

What exactly do we mean by "traditional call stack behavior"? Harrison identified tail-call optimization and escaping continuations as problematic for his approach. Any calling sequence that cannot be

Figure 5.3 Interprocedural dependency using stack reachability.

```
(define z 0)
(define (f) (g))
(define (g) (h))
(define (h) (set! z 123))

(f)
```

modeled using a standard stack mechanism must be disallowed because it destroys the analysis. This means that first-class escaping continuations that restore previously popped stack frames, as can be introduced by `call/cc`, must be disallowed. We will tackle the problem of tail-call optimization by using continuation marks (§5.4.4).

Harrison's procedure strings are in the realm of the concrete semantics of a program. Because they form an infinite set, they are not useful for analysis. The abstract interpretation of procedure strings are called *stack configurations* in Harrison's analysis, and they form a sound approximation of procedure strings at compile time.

5.4.2 Stack reachability

To make use of Harrison's dependence principle, we need to be able to inspect all the possible stack configurations that may arise at runtime while resources are read or written. To see how we can go about extracting interprocedural dependence information from these stack configurations, it might be useful to rephrase Harrison's dependence principle in terms of procedures instead of resources (still assuming proper call stack behavior).

Definition 5.4. (Interprocedural dependence via stack reachability) A procedure depends on all the resources it reads/writes directly and transitively on all of the resources read/written by its callees.

In order to safely inspect stack configurations when resources are read and written, we will build a finite-state abstract machine. From this abstract interpretation we can then conservatively bound the resources read and written by each procedure.

Example 5.5. The procedure `h` in the program in Figure 5.3 assigns the variable `z`. When this assignment occurs, frames for procedures `f`, `g`, and `h` are live on the stack. Hence `f`, `g`, and `h` all have a write dependency on variable `z`.

We still need to tidy up some loose ends concerning the traditional call-stack behavior that must be obeyed for the analysis to work.

- In the presence of proper tail-call optimization, tail calls do not push frames on the stack, invalidating the approach. Just as in the concrete case, abstract interpretations of tail-call-optimized semantics more efficiently use the stack resulting in higher precision [18]. Therefore it is worthwhile to preserve this property in the analysis. To solve the problem of tail-call optimization,

continuation marks are used. Continuation marks are annotations attached to frames (continuations) on the stack, ensuring that the necessary information on callers and calling contexts can still be recorded.

- Any calling sequence that cannot be modeled using a standard stack mechanism must be disallowed. This means that first-class escaping continuations that restore previously popped stack frames, as can be introduced by `call/cc`, are disallowed[2].

We will explore the idea of continuation marks further when we develop IPDA in the next few sections.

5.4.3 State machine

We implement the abstract analysis as an abstract state machine that models the concrete semantics of Scheme programs. It does not, however, accept the full Scheme language as input. Instead, ANF with the addition of a single imperative form `set!` is used as the input language for the analysis (Figure 3.2 on page 20). The reason ANF is a suitable intermediate form is because we want to abstractly model the stack to capture the control flow of the original source program, which is preserved when converting to ANF.

Our abstract state machine is an abstraction of a concrete state machine that is an interpreter for ANF. Because the concrete and the abstract semantics are structurally similar, *we will present the abstract semantics* because the abstract interpretation actually drives our analysis. We will note any important differences between the concrete and the abstract that might exist as we go along. We also will not provide the abstraction mappings from the concrete to the abstract domains (mapping α from §4.2). They are straightforward and should not pose any particular problem in understanding how the analysis works.

5.4.4 State space

Figures 5.4 and 5.5 show the concrete and abstract state spaces for the semantics. We will now discuss the entities of these domains that make up the states, while giving examples to better illustrate the concepts used.

In what follows, the notation $f[x \to y]$ means "the function f, except when applied to x yields value y". Abstract values can be joined through the join operator $\sqcup$: $\langle u_1, \ldots, u_n \rangle \sqcup \langle v_1, \ldots, v_n \rangle = \langle \{\{u_1, \ldots, u_n\} \cup \{v_1, \ldots, v_n\}\} \rangle$.

Timestamps

Every state contains a timestamp that serves the same purpose as contours in k-CFA; that is, they introduce a notion of contexts in the analysis between which we can distinguish.

In the concrete domain we have an infinite of set of times $Time$ that increases monotonically during interpretation (take for example $Time = \mathbb{N}$). In the abstract case we have compress this infinite set

[2] In principle we could handle some more advanced mechanisms like non-local exits through `call/cc` since stack behavior remains predictable (non-local exits pop more than just the top frame off the stack) but this would require extra discipline by the programmer or extra checks by the compiler whenever `call/cc` appears in a program. Therefore it is better if we disallow `call/cc` and other similar constructs that can be expressed through `call/cc` (like try/catch/throw for example) altogether.

Figure 5.4 Concrete state space.

$$
\begin{array}{rcll}
\varsigma & \in & State & = EvalHead + EvalTail + ApplyFun + ApplyKont \\
 & & EvalHead & = \mathsf{Exp} \times Benv \times Store \times Kont \times Time \\
 & & EvalTail & = \mathsf{Exp} \times Benv \times Store \times RetPoint \times Time \\
 & & ApplyFun & = Clo \times Val^* \times Store \times RetPoint \times Time \\
 & & ApplyKont & = Kont \times Val \times Store \times Time \\
 & & & \\
\beta & \in & Benv & = \mathsf{Var} \to Addr \\
\sigma & \in & Store & = Addr \to Val \\
 & & & \\
a & \in & Addr & = Bind + RetPoint \\
b & \in & Bind & = \mathsf{Var} \times Time \\
 & & & \\
v & \in & Val & = \mathsf{Quo} + \mathsf{Prim} + Clo + Kont \\
\chi & \in & Clo & = \mathsf{Lam} \times Benv \\
\kappa & \in & Kont & = \mathsf{Var} \times \mathsf{Exp} \times Benv \times RetPoint \times Mark \\
 & & & \\
rp & \in & RetPoint & = \text{a set of addresses for continuations} \\
m & \in & Mark & = \text{a set of stack-frame annotations} \\
t & \in & Time & = \text{an infinite set of times}
\end{array}
$$

Figure 5.5 Abstract state space.

$$
\begin{array}{rcll}
\hat{\varsigma} & \in & \widehat{State} & = \widehat{EvalHead} + \widehat{EvalTail} + \widehat{ApplyFun} + \widehat{ApplyKont} \\
 & & \widehat{EvalHead} & = \mathsf{Exp} \times \widehat{Benv} \times \widehat{Store} \times \widehat{Kont} \times \widehat{Time} \\
 & & \widehat{EvalTail} & = \mathsf{Exp} \times \widehat{Benv} \times \widehat{Store} \times \widehat{RetPoint} \times \widehat{Time} \\
 & & \widehat{ApplyFun} & = \widehat{Clo} \times \widehat{Val}^* \times \widehat{Store} \times \widehat{RetPoint} \times \widehat{Time} \\
 & & \widehat{ApplyKont} & = \widehat{Kont} \times \widehat{Val} \times \widehat{Store} \times \widehat{Time} \\
 & & & \\
\hat{\beta} & \in & \widehat{Benv} & = \mathsf{Var} \to \widehat{Addr} \\
\hat{\sigma} & \in & \widehat{Store} & = \widehat{Addr} \to \widehat{Val} \\
 & & & \\
\hat{a} & \in & \widehat{Addr} & = \widehat{Bind} + \widehat{RetPoint} \\
\hat{b} & \in & \widehat{Bind} & = \mathsf{Var} \times \widehat{Time} \\
 & & & \\
\hat{v} & \in & \widehat{Val} & = \mathcal{P}(\mathsf{Quo} + \mathsf{Prim} + \widehat{Clo} + \widehat{Kont}) \\
\hat{\chi} & \in & \widehat{Clo} & = \mathsf{Lam} \times \widehat{Benv} \\
\hat{\kappa} & \in & \widehat{Kont} & = \mathsf{Var} \times \mathsf{Exp} \times \widehat{Benv} \times \widehat{RetPoint} \times \widehat{Mark} \\
 & & & \\
\hat{rp} & \in & \widehat{RetPoint} & = \mathsf{Lam} \times \widehat{Time} \\
\hat{m} & \in & \widehat{Mark} & = \mathsf{Proc} \times \mathsf{App} \\
\hat{t} & \in & \widehat{Time} & = \text{list of } k \text{ last call sites}
\end{array}
$$

Figure 5.6 Program with an infinite number of bindings to `x`

```
(letrec ((f (lambda (x)
              (f (+ x 1))))
  (f 0))
```

Figure 5.7 Double-recursive Fibonacci function in ANF

```
(letrec ((fib (lambda (n)
                (let ((#p0 (<35 n 2)))
                  (if #p0
                      n
                      (let ((#p1 (-41 n 1)))
                        (let ((#p2 (fib46 #p1)))
                          (let ((#p3 (-52 n 2)))
                            (let ((#p4 (fib57 #p3)))
                              (+63 #p2 #p4)))))))))))
  (fib78 2))
```

into a finite set $\widehat{Time}$. Elements from $\widehat{Time}$ represent a bounded amount of evaluation history. The typical example for the construction of a set of abstract times are the contours of k-CFA. Every time the abstract interpreter encounters an application site, we `cons` the label of the application onto a list and keep at most the first k elements. In the remainder of the analysis this will be the way we construct abstract timestamps.

Bindings

Bindings play the role of pointers or addresses. In order to work with multiple bindings of the same variable, a binding is the name of the variable coupled to a timestamp. The role of the timestamp is to provide a source of freshness when allocating bindings. The variable-time pair $(u, \hat{t})$ means "variable u bound at time $\hat{t}$".

Example 5.6. (Concrete bindings) In the program in Figure 5.6 there are an infinite number of concrete bindings to `x`, each made at a different time and each associated with a different value.

Since both the set of abstract times and the number of applications in a given program are finite, it is possible that multiple concrete bindings of the same variable map onto the same abstract binding.

We assume variable collision cannot occur due to two or more variables having the same name. The analysis expects unique variable names, for example through α-conversion.

Example 5.7. (Abstract bindings) Suppose we have a double-recursive Fibonacci function with application sites labeled as in Figure 5.7. In our analysis we are tracking the bindings of variable `n` using the k most recent call sites as abstract time. It is obvious that when `n` as formal parameter gets bound for the first time, the abstract time will be $()$ for $k = 0$ and (78) for all other values of k.

When we take $k = 4$ for example, some of the possible timestamps for bindings of `n` are (78), $(46, 41, 35, 78)$, $(46, 41, 35, 46)$, $(57, 52, 35, 46)$, $(46, 41, 35, 46)$, $(46, 41, 35, 57)$, $(57, 52, 63, 35)$, and $(57, 52, 63, 63)$.

Figure 5.8 A closure captures the binding environment at the time of its creation.

```
(let ((x (f20 3)))
  (lambda42 (n) (+ x n)))
```

Every timestamp allows us to trace a path through the code to see how the interpreter arrived at the binding site of `n`.

Binding-factored environment

Variables are mapped to values in two stages. First, a variable maps onto a binding through a *binding environment* $\hat{\beta}$. Next, a binding maps to a value through the store $\hat{\sigma}$. A variable lookup effectively is the composition of these two functions and the value of variable u is given by $\hat{\sigma}(\hat{\beta}(u))$.

Mapping variables to values in this fashion allows us to reason at the level of individual bindings and expresses the fact that during interpretation multiple bindings to the same variable can be simultaneously live. This simplifies the semantics of mutation in the analysis considerably, and at the same time avoids the need for machinery for dealing with recursion directly.

When using a binding-factored environment – as opposed to an unfactored environment where a variable points directly to its value – the part of the store that contains bindings acts as the *heap* and bindings as addresses in that heap. We can define a function $\widehat{succ} : \widehat{State} \to \widehat{Time}$ that, given a state, computes the next time. In the case of 0-CFA, the set $\widehat{Times}$ is a singleton $\{()\}$.

Given a store $\hat{\sigma}$ for which $\hat{\sigma}(\hat{a}) = \hat{u}$ and a mapping $\hat{a} \to \hat{v}$ we would like to add, we can update $\hat{\sigma}$ in two ways:

- We can *join* the existing values at address $\hat{a}$ with $\hat{v}$: the resulting store is $\hat{\sigma}' = \hat{\sigma} \sqcup [\hat{a} \to \hat{v}]$ and $\hat{\sigma}'(\hat{a}) = \hat{u} \sqcup \hat{v}$. We call this type of update simply a join or a *weak update*.
- We can replace the existing values at address $\hat{a}$ with $\hat{v}$: the resulting store is $\hat{\sigma}' = \hat{\sigma}[\hat{a} \to \hat{v}]$ and $\hat{\sigma}'(\hat{a}) = \hat{v}$. This type of update is called a *strong update*, or an update via shadowing.

In case $\hat{\sigma}(\hat{a}) = \emptyset$ to begin with, the type of update for adding mapping $\hat{a} \to \hat{v}$ is immaterial and the result identical.

Closures

A closure is a pair of a lambda expression and a binding environment that is captured at the time of its creation. Closures are created when lambdas are evaluated.

Example 5.8. (Closures) The result after abstract execution of the program in Figure 5.8 is the abstract closure $\hat{\chi} = (\lambda_{42}, [\mathtt{x} \to (\mathtt{x}, (20))])$.

Return points

Return points are addresses in the store that point to continuations. The portion of the store that holds return points can be considered as the *stack*. Because return points are needed when applications are involved, we encode abstract return points as lambda expressions. Example 5.9 shows how this works.

Figure 5.9 Creation of continuations for `let` expressions.

```
(let24 ((f (lambda17 (n) ...)))
  (let22 ((g (f 3)))
    (h g)))
```

Just as was the case with bindings, we control the polyvariance of the stack by adding the abstract timestamp at which the return point was created.

Continuations

A continuation receives a value and assigns it to the variable it contains. It also consists of an expression to evaluate, the binding environment in which to evaluate the expression, and a pointer to the continuation beneath it. Lastly, it also contains a mark that will hold annotations. We will explain marks and annotations in more detail later on.

The analysis does not use procedure strings to model the stack, as was the case in Harrison's analysis. Instead an "actual" stack is modeled in the store by chaining continuations through return points. Walking a stack of continuations is crucial if we want to apply Harrison's principle.

Due to the nature of ANF, the only place where our interpreter has to "remember" something is when it has evaluate a let-expression that involves binding an application. It has to remember that after evaluating the body of the procedure that is invoked it has to continue with the body of the `let`. For this purpose a continuation is set up with the name of the let-bound variable and the body of the `let` as the expression. The interpreter then proceeds to evaluate the let-bound application with the freshly created continuation as a return point.

There is one special continuation, called the *halt continuation*, that signifies the end of a program execution. We will denote the halt continuation as $\hat{\kappa}_0$, and the return point pointing to the halt continuation as $\hat{rp}_0$. The value that the halt continuation receives is the result of the program. It has no meaningful "next" expression. The halt continuation is also the initial continuation supplied to the interpreter at the start of evaluation.

Example 5.9. (Abstract continuations) There are two let expressions in the program of Figure 5.9: an outer let expression binding a lambda and an inner let expression binding the result of an application.

When the outer `let` is evaluated by our abstract interpreter, the lambda expression is evaluated to the closure $\hat{\chi}_{17} = (\lambda_{17}, [])$ and bound to `f`. The inner `let` binds the result of an application, so a continuation $\hat{\kappa}_{22} = (\texttt{g}, (\texttt{h g}), [\texttt{f} \rightarrow (\texttt{f}, ())], \hat{rp}_0, [])$ is created, as well as a return point $\hat{rp}_{22} = (\lambda_{17}, ())$ pointing to $\hat{\chi}_{17}$. $\lambda_{17}$is used for $\hat{rp}_{22}$ because `f` evaluates to closure $\hat{\chi}_{17}$ which contains this lambda expression.

Continuation marks

Continuation marks are annotations attached to stack frames and serve to record procedure activations in the presence of tail-call optimization. Every time a continuation acts as a return point, it is marked with the current procedure. Using marks on continuations, it is clear that every procedure activation will be recorded, even if no new continuation is created for tail-called procedures. In fact, tail-called

procedures share the same continuation (or continuations in the abstract case) in our tail-call optimized semantics.

Tail positions allowing applications in our ANF language (Figure 3.2) are the body of a `lambda`, `let`, and `letrec`, and the consequent and alternate expression of an `if`. Non-tail application sites are only allowed when they are let-bound, i.e. when the application appears as a value inside a let expression. As we have seen, in the latter case the interpreter will create a fresh continuation to continue with the body of the `let` after procedure invocation.

Example 5.10. (Continuation marks) In the program in Figure 5.1, three continuations are created/used during interpretation: $\hat{\kappa}_0$ (halt), $\hat{\kappa}_{20}$ when evaluating `(writez)` and $\hat{\kappa}_{24}$ when evaluating `(readz)`.

We will now show how marks are attached to continuations in the relevant evaluation states. For clarity, and because there is no ambiguity, we will not include contexts for bindings and marks. When evaluating `(set! ...)` in procedure `writez`, the stack is $\hat{\kappa}_{20} \rightarrow \hat{\kappa}_0$ and $\hat{\kappa}_{20}$ is marked with λ_{12}. When evaluating `z` in procedure `readz`, the stack is $\hat{\kappa}_{24} \rightarrow \hat{\kappa}_0$ and $\hat{\kappa}_{24}$ is marked with λ_{16}. Finally, no new continuation is created for evaluating `(cons p1 p2)` and $\hat{\kappa}_0$ is marked with `cons`.

Just as we made stack and heap allocations (return points and bindings) context-sensitive by coupling them with timestamps that reflect the k latest call sites, we can make the analysis context-sensitive with respect to dependences by enriching our context marks with the application site that gives rise to the mark itself. More clearly, if procedure $proc$ flows to application $a = (f\ x_1 \ldots x_n)$, then the corresponding mark will be $(proc, a)$.

5.4.5 Atomic expression evaluator

Our interpreter will make use of an atomic expression evaluator when evaluating variable references, literals, primitives and lambda expressions. These entities are exactly what we consider to be *atomic expressions*, leading to the following definition.

Definition 5.11. (Atomic expression) An *atomic expression* is one whose evaluation must terminate (i.e. it yields a value immediately) and which produces no side effects.

In our semantics we can now introduce an atomic evaluator function $\hat{\varepsilon} : \mathsf{Arg} \times \widehat{Benv} \times \widehat{Store} \rightarrow \widehat{Val}$ which is defined as follows:

$$\begin{cases} \hat{\varepsilon}(\lambda, \hat{\beta}, \hat{\sigma}) = (\lambda, \hat{\beta}) & \text{(closure)} \\ \hat{\varepsilon}(u, \hat{\beta}, \hat{\sigma}) = \hat{\sigma}(\hat{\beta}(u)) & \text{(value lookup)} \\ \hat{\varepsilon}(q, \hat{\beta}, \hat{\sigma}) = q & \text{(constant)} \\ \hat{\varepsilon}(p, \hat{\beta}, \hat{\sigma}) = p & \text{(primitive)} \end{cases}$$

One of the major advantages of using ANF as an intermediary form during analysis really shows when looking at how atomic expressions help simplify the operation of the interpreter. When dealing with function application for example, ANF ensures that all operands are atomic expressions, greatly simplifying the semantics. When confronted with an application site with some list of arguments, the

interpreter can simply map the atomic evaluator function over the arguments, before actually applying the operator. Compare this to a "regular" interpreter, where the evaluation of non-trivial arguments requires pushing the stack by calling the (full) evaluator function recursively over the list of arguments.

The same advantages of atomic expressions are encountered when the interpreter has to deal with conditions of if-expressions, values inside `set!` statements and letrec-bindings, and the values that can be bound in let expressions that are either applications with atomic arguments or atomic expressions themselves.

5.4.6 State transitions

Our state machine features four types of states: two *evaluation states* and two *application states*. Every state performs some specific task. The evaluation states take care of evaluating expressions. The application states apply functions and continuations. Every *concrete* state gives rise to a single successor state and is deterministic. An *abstract* state can spawn multiple successor state and is *non-deterministic*.

It is easy to see why the abstract state machine is necessarily non-deterministic: our abstract interpreter deals with abstract values. Abstract values representing multiple concrete values are bound, passed to and returned from procedures, and so on. Also, when we lookup procedures or continuations in the store, multiple concrete entities may be returned that must be applied. An additional consequence is that when evaluating a conditional, the abstract interpreter may choose to follow multiple branches when it has not enough information to decide upon following a specific branch (e.g. when the condition of an if expression evaluates to abstract value $\langle \#\mathtt{t}, \#\mathtt{f} \rangle$).

We will now discuss the behavior of the four abstract states and note most of the differences with the concrete states.

EvalTail

$\widehat{EvalTail}$ evaluates expressions that are in tail position. Due to the nature of our input language, the possible tail positions are the bodies of lambda expressions and let expressions. The behavior of $(x, \hat{\beta}, \hat{\sigma}, \hat{rp}, \hat{t}) \in \widehat{EvalTail}$ can be broken down in different cases that follow the syntactic type of expression x.

- For a lambda expression, reference or a literal, we perform an atomic evaluation of x to obtain value $\hat{v} = \hat{\varepsilon}(x, \hat{\beta}, \hat{\sigma})$. Next, we look up the continuations in the store associated with the current return point: $\hat{\sigma}(\hat{rp}) = \langle \hat{\kappa}_1, \ldots, \hat{\kappa}_n \rangle$. For each continuation $\hat{\kappa}$ in that list, we create a successor state $(\hat{\kappa}, \hat{v}, \hat{\sigma}, \hat{t}) \in \widehat{ApplyKont}$. (In the concrete case, there would only be one continuation associated with rp, and hence only one successor state.)

- In case of a tail call, we are dealing with an application $e_{app} = (f\ x_1 \ldots x_n)$. First we update the timestamp using the tag n associated with e_{app}: $\hat{t}' =$ first k elements of $(n + \hat{t})$. Next we atomically evaluate operator and operands. If the values for the operands are $\hat{vs}_{rands} = \hat{\varepsilon}(\langle x_1, \ldots, x_n \rangle, \hat{\beta}, \hat{\sigma})$, then for every value $\hat{v}_{rator} \in \hat{\varepsilon}(f, \hat{\beta}, \hat{\sigma})$ we distinguish two subcases:

  - if $\hat{v}_{rator} = (lam, \hat{\beta}_\chi) \in \widehat{Clo}$ then we mark every continuation in the list $\hat{\sigma}(\hat{rp})$ with (lam, e_{app}), and we add $(\hat{v}_{rator}, \hat{vs}_{rands}, \sigma, rp, t') \in \widehat{ApplyFun}$ to the list of successor states,

  - else $\hat{v}_{rator} \in \mathsf{Prim}$, we evaluate the primitive passing arguments $\hat{vs}_{rands}$ yielding value $\hat{v}_{app}$, and for every continuation $\hat{\kappa} \in \hat{\sigma}(\hat{rp})$ we mark $\hat{\kappa}$ with $(\hat{v}_{rator}, e_{app})$ and we add $(\hat{\kappa}, \hat{v}_{app}, \hat{\sigma}, \hat{t}') \in \widehat{ApplyKont}$ to the list of successor states.

  (In the concrete case we only have one evaluated operator and one continuation for return point rp.)

- If we encounter a binding mutation through $(\texttt{set!}\ u\ x)$ then we obtain address $\hat{a} = \hat{\beta}(u)$ and value $\hat{v} = \hat{\varepsilon}(x, \hat{\beta}, \hat{\sigma})$. Next, we update the store at address $\hat{a}$ with $\hat{v}$: $\hat{\sigma}' = \hat{\sigma} \sqcup [\hat{a} \to \hat{v}]$. We return $(?, \hat{\beta}, \hat{\sigma}', \hat{rp}, \hat{t}) \in \widehat{EvalTail}$ as successor state, where $?$ stands for an unspecified value. (In the concrete case we would not use joining to update the store, but simply overwrite the existing value in σ associated with a to obtain σ'.)

- For a let expression $(let\ ((u\ e_{app}))\ e_{body})$ binding application $e_{app} = (f\ x_1 \ldots x_n)$, we create continuation $\hat{\kappa} = (u, e_{body}, \hat{\beta}, \hat{rp})$ that will evaluate the body of our let after assigning the result of e_{app} to u. The successor state is $(e_{app}, \hat{\beta}, \hat{\sigma}, \hat{\kappa}, \hat{t}) \in \widehat{EvalHead}$.

- For a non-application binding let expression $(\texttt{let}\ ((u\ x))\ e_{body})$ we atomically evaluate x to obtain value $\hat{v} = \hat{\varepsilon}(u, \hat{\beta}, \hat{\sigma})$. Then we extend the binding environment $\hat{\beta}$ with binding $\hat{b} = (u, \hat{t})$ so that $\hat{\beta}' = \hat{\beta}[u \to \hat{b}]$, while allocating value $\hat{v}$ on the heap at address $\hat{b}$: $\hat{\sigma}' = \hat{\sigma} \sqcup [\hat{b} \to \hat{v}]$. As successor state we have $(e_{body}, \hat{\beta}', \hat{\sigma}', \hat{rp}, \hat{t}) \in \widehat{EvalTail}$. (The concrete interpreter would not use a join to update store σ but would simply overwrite any existing value at address b.)

- If we have a recursive function u expressed as $(\texttt{letrec}\ ((u\ lam))\ e_{body})$ then we first create a new binding environment in which u points to binding $\hat{b} = (u, \hat{t})$: $\hat{\beta}' = \hat{\beta}[u \to \hat{b}]$. We use $\hat{\beta}'$ to atomically evaluate lam so that we get value $\hat{v} = \hat{\varepsilon}(lam, \hat{\beta}', \hat{\sigma})$. We extend store $\hat{\sigma}$ to allocate value $\hat{v}$ at address $\hat{b}$ obtaining $\hat{\sigma}' = \hat{\sigma} \sqcup [\hat{b} \to \hat{v}]$. Finally, our successor state $(e_{body}, \hat{\beta}', \hat{\sigma}', \hat{rp}, \hat{t}) \in \widehat{EvalTail}$ will evaluate the body of our `letrec`.

- Our last case deals with a conditional expression of the form $(\texttt{if}\ x_{cond}\ e_{cons}\ e_{alt})$. If $\hat{v} = \hat{\varepsilon}(x, \hat{\beta}, \hat{\sigma})$, the result of atomically evaluating the condition, then we distinguish amongst the following subcases:

  - $\hat{v} = \langle \#\texttt{f} \rangle$: the successor state is $(e_{alt}, \hat{\beta}, \hat{\sigma}, \hat{rp}, \hat{t}) \in \widehat{EvalTail}$,
  - $\#\texttt{f} \notin \hat{v}$ and $\hat{v} \neq \emptyset$: the successor state is $(e_{cons}, \hat{\beta}, \hat{\sigma}, \hat{rp}, \hat{t}) \in \widehat{EvalTail}$
  - else we return the two successor states from the above cases.

  (In the concrete case, we will have either $v = \#\texttt{t}$ or $v = \#\texttt{f}$ and only the two first subcases above with a single successor state are applicable.)

EvalHead

$\widehat{EvalHead}$ evaluates expressions that are in non-tail position. Due to the nature of our input language and the construction of our state machine, the only possible non-tail position we need to consider are application sites that are let-bound.

Suppose $(e_{app}, \hat{\beta}, \hat{\sigma}, \hat{\kappa}, \hat{t}) \in \widehat{EvalHead}$ and $e_{app} = (f\ x_1 \ldots x_n)$. As was the case with tail-calls, we update the timestamp to obtain $\hat{t}' = prefix(n+\hat{t}, k)$ using the tag n associated with e_{app} and we atomically evaluate operator and operands. If the values for the operands are $\hat{vs}_{rands} = \hat{\varepsilon}(\langle x_1, \ldots, x_n \rangle, \hat{\beta}, \hat{\sigma})$, then for every value $\hat{v}_{rator} \in \hat{\varepsilon}(f, \hat{\beta}, \hat{\sigma})$ we distinguish two cases:

- if $\hat{v}_{rator} = (lam, \hat{\beta}_\chi) \in \widehat{Clo}$ then we mark every continuation in the list $\hat{\sigma}(\hat{rp})$ with (lam, e_{app}). We allocate return point $\hat{rp} = (lam, \hat{t})$ on the stack by inserting it in our store $\hat{\sigma}' = \hat{\sigma}[\hat{rp} \to \langle \hat{\kappa} \rangle]$. We can now add $(\hat{v}_{rator}, \hat{vs}_{rands}, \hat{\sigma}', \hat{rp}, \hat{t}') \in \widehat{ApplyFun}$ to the list of successor states,
- else $\hat{v}_{rator} \in$ Prim, we evaluate the primitive passing $\hat{vs}_{rands}$ as arguments, yielding value $\hat{v}_{app}$. For every continuation $\hat{\kappa} \in \hat{\sigma}(\hat{rp})$ we mark $\hat{\kappa}$ with $(\hat{v}_{rator}, e_{app})$ and we add $(\hat{\kappa}, \hat{v}_{app}, \hat{\sigma}, \hat{t}') \in \widehat{ApplyKont}$ to the list of successor states.

ApplyFun

ApplyFun applies a lambda to some (possibly empty) list of arguments and then moves on to evaluate the body of that lambda.

Suppose that $(\hat{\chi}, \hat{vs}_{rands}, \hat{\sigma}, \hat{rp}, \hat{t}) \in \widehat{ApplyFun}$ with closure $\hat{\chi} = (lam, \hat{\beta}_{lam})$ and lambda expression $lam = (\lambda(u_1 \ldots u_n)\ e_{body})$. Starting with $\hat{\beta}' = \hat{\beta}$ and $\hat{\sigma}' = \hat{\sigma}$, we allocate every argument $\hat{v}_i \in \hat{vs}_{rands}$ on the heap by creating a binding $\hat{b} = (u_i, \hat{t})$, extending the binding environment $\hat{\beta}' = \hat{\beta}'[u_i \to \hat{b}]$ and inserting $\hat{v}_i$ into the store $\hat{\sigma}' = \hat{\sigma}' \sqcup [\hat{b} \to \hat{v}_i]$. (In the concrete case we do not insert into the store using a join, but by overwriting existing values that may exist at the addresses of the arguments.) The successor state is $(e_{body}, \hat{\beta}', \hat{\sigma}', \hat{rp}, \hat{t}) \in \widehat{EvalTail}$.

ApplyKont

ApplyKont assigns a value to a name and continues evaluation at a return point. The halt continuation is treated specially and effectively halts the interpreter in the concrete case, and terminates a branch of exploration in the abstract case.

Suppose that $(\hat{\kappa}, \hat{v}, \hat{\sigma}, \hat{t}) \in \widehat{ApplyKont}$ with continuation $\hat{\kappa} = (u, e, \hat{\beta}_\kappa, \hat{rp}, \hat{m})$. The behavior then is to extend binding environment $\hat{\beta}_\kappa$ so that identifier u points to binding $\hat{b} = (u, \hat{t})$: $\hat{\beta}' = \hat{\beta}_\kappa[u \to \hat{b}]$. We also insert value $\hat{v}$ into the store so that $\hat{\sigma}' = \hat{\sigma} \sqcup [\hat{b} \to \hat{v}]$ and we return $(e, \hat{\beta}', \hat{\sigma}', \hat{rp}, \hat{t}) \in \widehat{EvalTail}$ as the successor state. (In the concrete case we do not insert value v into the store using a join, but by overwriting any existing value at address b.)

5.4.7 State exploration

We have now discussed the behavior of all the states involved, and have in fact defined a concrete semantic transition function $(\Rightarrow) : State \to State$ and abstract semantic relation $(\rightsquigarrow) : \widehat{State} \times \widehat{State}$ in a case-by-case fashion. What is still missing is a starting point. If we want to analyze program e, we need to inject it into the state space and we do so by using injector function $\hat{\mathcal{I}} : \text{Exp} \to \widehat{State}$. We can define $\hat{\mathcal{I}}$ as $\hat{\mathcal{I}}(e) = (e, \hat{\beta}_0, \hat{\sigma}_0, \hat{rp}_0, \hat{t}_0) \in \widehat{EvalTail}$, where $\hat{\beta}_0$ and $\hat{\sigma}_0$ are the empty binding environment and store respectively, $\hat{rp}_0$ is the return point for $\hat{\kappa}_0$ (the halt continuation), and $\hat{t}_0 = ()$ is the initial timestamp.

From an initial state we then trace out the set of all visitable states. A naive exploration algorithm would keep a work set and a set of seen states. It operates by taking a state from the work set. If this state is already encountered (i.e. it is a member of the set of seen states) the algorithm continues with the next state from the work set. Else the state is added to the set of seen states and its successors are computed and added to the work set. The algorithm terminates when the work set is empty. The set of seen states $\hat{\mathcal{V}} = \{\hat{\varsigma} | \hat{\mathcal{I}}(e) \rightsquigarrow^* \hat{\varsigma}\}$ is then the set of states visited by the interpreter. In the abstract case we have

The above algorithm is naive because it is not realistic to implement state exploration as described above for any but the smallest input programs. There are improvements we can make to get faster and better results out of the analysis (§5.5).

5.4.8 Computing data dependence

When we have a set of explored states $\hat{\mathcal{V}}$ we can proceed to the actual extraction of information to create our dependency graph that relates abstract procedures to abstract resources. Abstract procedures are the marks collected by continuations that form the call stack, i.e. they are of the form $\hat{m} = \mathsf{Proc} \times \mathsf{App}$. Abstract resources are sets of mutable concrete resources, in our case sets of mutable heap addresses (bindings for mutable variables) of the form $\hat{b} = \mathsf{Var} \times \widehat{Time}$.

Note that we are dealing with two different types of contexts here:

- the context-sensitivity of the heap determined by the last k call sites,
- the context-sensitivity of dependences determined by the actual call site.

As Might and Prabhu remark in [19], the choice of making abstract procedures context-sensitive can be made *independent* of the choice of making abstract resources context-sensitive. Thus it is possible to have a context-sensitive dependence analysis while still having a context-insensitive abstract interpreter for example. In the same paper the authors also discuss some alternatives on how to make the dependence analysis context-sensitive. One could also synchronize the context-sensitivity of abstract procedures with the context-sensitivity of the stack for example, so that $\widehat{Mark} = \mathcal{P}(\mathsf{Lam} \times \widehat{RetPoint})$. Another option could be to synchronize it with the context-sensitivity of the heap, resulting in $\widehat{Mark} = \mathcal{P}(\mathsf{Lam} \times \widehat{Time})$.

To actually compute our dependence graph we inspect the stack configuration of every visited state $\hat{\varsigma} \in \hat{\mathcal{V}}$. The state stack can then be traversed to aggregate the continuation marks. Analyzing these continuation marks together with the read and write dependencies per abstract state yields the dependence graph.

First of all we need the following ingredients dealing with the stack configuration per state.

- We need a function $\hat{\mathcal{S}}_{\mathcal{R}} : \widehat{State} \to \mathcal{P}(\widehat{Cont})$ that, given a state $\hat{\varsigma}$ returns all its root continuations. For $\hat{\varsigma} \in \widehat{EvalTail}$ or $\hat{\varsigma} \in \widehat{ApplyFun}$ with store $\hat{\sigma}_{\hat{\varsigma}}$ and return point $\hat{rp}$, the root continuations are those in the store pointed to by address $\hat{rp}$:

$$\hat{\mathcal{S}}_{\mathcal{R}}(\hat{\varsigma}) = \hat{\sigma}_{\hat{\varsigma}}(\hat{rp})$$

For $\hat{\varsigma} \in \widehat{EvalHead}$ or $\hat{\varsigma} \in \widehat{ApplyKont}$ with continuation $\hat{\kappa}$ we simply take

$$\hat{\mathcal{S}}_{\mathcal{R}}(\hat{\varsigma}) = \langle \hat{\kappa} \rangle$$

- Starting from a set of root continuations we can "walk the stack" for any given state by exploiting pointers between continuations created during abstract interpretation. For a state $\hat{\varsigma}$ with store $\hat{\sigma}_{\hat{\varsigma}}$ the adjacency relation on continuations $(\rightarrow_{\hat{\varsigma}}) \subseteq \widehat{Kont} \times \widehat{Kont}$ is given by

$$\hat{\kappa} = (u, e, \hat{\beta}, \hat{rp}, \hat{m}) \rightarrow_{\hat{\varsigma}} \hat{\kappa}' \iff \hat{\kappa}' \in \hat{\sigma}_{\hat{\varsigma}}(\hat{rp})$$

- We can then construct a reachability function $\hat{\mathcal{S}}^* : \widehat{State} \rightarrow \mathcal{P}(\widehat{Cont})$ to find all the continuations reachable in a particular state by taking the transitive closure of $\rightarrow_{\hat{\varsigma}}$ applied to the root continuations of that state:

$$\hat{\mathcal{S}}^*(\hat{\varsigma}) = \{\hat{\kappa} | \hat{\kappa}_{\hat{\varsigma}} \rightarrow^*_{\hat{\varsigma}} \hat{\kappa} \wedge \hat{\kappa}_{\hat{\varsigma}} \in \hat{\mathcal{S}}_{\mathcal{R}}(\hat{\varsigma})\}$$

- Now that we can traverse all continuations reachable in a state $\hat{\varsigma}$, we can collect all the marks attached to those continuations as we go along, using the aggregate mark function $\hat{\mathcal{M}} : \widehat{State} \rightarrow \mathcal{P}(\widehat{Mark})$, defined as

$$\hat{\mathcal{M}}(\hat{\varsigma}) = \bigcup_{\hat{\kappa} \in \hat{\mathcal{S}}^*(\hat{\varsigma})} \hat{m}_{\hat{\kappa}}$$

We also need support for determining read and write dependencies per state.

- Let $\hat{\mathcal{A}} : \widehat{Benv} \rightarrow \mathsf{Exp}^* \rightarrow \mathcal{P}(\widehat{Addr})$ be the function that yields the addresses immediately read by expressions. Then for a variable u we have

$$\hat{\mathcal{A}}(\hat{\beta})\,\langle u \rangle = \{\hat{\beta}(u)\}$$

and for all other expressions $e \notin \mathsf{Var}$

$$\hat{\mathcal{A}}(\hat{\beta})\,\langle e \rangle = \emptyset$$

By extension we also define

$$\hat{\mathcal{A}}(\hat{\beta})\,\langle\rangle = \emptyset$$

and

$$\hat{\mathcal{A}}(\hat{\beta})\,\langle e_1, \ldots, e_n \rangle = \hat{\mathcal{A}}(\hat{\beta})(e_1) \cup \cdots \cup \hat{\mathcal{A}}(\hat{\beta})(e_n)$$

- For an evaluation state $\hat{\varsigma}$ with expression $e_{\hat{\varsigma}}$ and binding environment $\hat{\beta}_{\hat{\varsigma}}$, we can compute the set of abstract addresses read by that state. We do so by constructing a function $\hat{\mathcal{R}} : \widehat{State} \rightarrow \mathcal{P}(\widehat{Addr})$ that we define following the type of expression.
  - If $e_{\hat{\varsigma}} = x \in \mathsf{Arg}$ then $\hat{\mathcal{R}}(\hat{\varsigma}) = \hat{\mathcal{A}}(\hat{\beta}_{\hat{\varsigma}})\,\langle x \rangle$, i.e. only if x is a variable an address is read. For literals, primitives and lambdas, $\hat{\mathcal{R}}(\hat{\varsigma}) = \emptyset$.
  - For a function application $e_{\hat{\varsigma}} = (f\ x_1 \ldots x_n)$, we have $\hat{\mathcal{R}}(\hat{\varsigma}) = \hat{\mathcal{A}}(\hat{\beta}_{\hat{\varsigma}})\,\langle f, x_1, \ldots, x_n \rangle$.
  - In case of a let expression $e_{\hat{\varsigma}} = (\texttt{let}\ ((u\ e_{value}))\ e_{body})$, we return the expressions immediately read by the value expression only, so $\hat{\mathcal{R}}(\hat{\varsigma}) = \hat{\mathcal{A}}(\hat{\beta}_{\hat{\varsigma}})\,\langle e_{value} \rangle$.

  - When defining a recursive function through $e_{\hat{\varsigma}} = (\texttt{letrec}\ ((u\ lam))\ e_{body})$, no variables are directly read and $\hat{\mathcal{R}}(\hat{\varsigma}) = \emptyset$.
  - For a variable mutation $e_{\hat{\varsigma}} = (\texttt{set!}\ u\ x)$, we have $\hat{\mathcal{R}}(\hat{\varsigma}) = \hat{\mathcal{A}}(\hat{\beta}_{\hat{\varsigma}})\ \langle x \rangle$.
  - For a conditional expression $e_{\hat{\varsigma}} = (\texttt{if}\ x_{cond}\ e_{cons}\ e_{alt})$, we return the expressions immediately read by the condition only, so $\hat{\mathcal{R}}(\hat{\varsigma}) = \hat{\mathcal{A}}(\hat{\beta}_{\hat{\varsigma}})\ \langle x_{cond} \rangle$.

- In a similar manner we define a function $\hat{\mathcal{W}} : \widehat{State} \to \mathcal{P}(\widehat{Addr})$ that computes the addresses written by an evaluation state $\hat{\varsigma}$ with expression $e_{\hat{\varsigma}}$ and binding environment $\hat{\beta}_{\hat{\varsigma}}$.

  - For a variable mutation $e_{\hat{\varsigma}} = (\texttt{set!}\ u\ x)$, we have $\hat{\mathcal{W}}(\hat{\varsigma}) = \hat{\mathcal{A}}(\hat{\beta}_{\hat{\varsigma}})\ \langle u \rangle = \{\hat{\beta}_{\hat{\varsigma}}(u)\}$.
  - For certain primitive procedures, including `set-car!`, `set-cdr!` and `vector-set!`, one of the operands (the first in the three primitives mentioned) represents an address that will be written. E.g. for $e_{\hat{\varsigma}} = (\texttt{set-car!}\ u\ x)$ we have $\hat{\mathcal{W}}(\hat{\varsigma}) = \{u\}$.
  - For all other inputs $\hat{\mathcal{W}}(\hat{\varsigma}) = \emptyset$.

Combining stack configurations and read/write dependencies per state, we can create the abstract resource dependence graph.

- For every state $\hat{\varsigma} \in \hat{\mathcal{V}}$ we determine $\hat{\mathcal{M}}(\hat{\varsigma})$, $\hat{\mathcal{R}}(\hat{\varsigma})$ and $\hat{\mathcal{W}}(\hat{\varsigma})$.

  - For every mark $\hat{m}_{\hat{\varsigma}}$ in $\hat{\mathcal{M}}(\hat{\varsigma})$ we add a vertex to the graph representing an abstract procedure invocation.
    * For every address $\hat{r}_{\hat{\varsigma}}$ in $\hat{\mathcal{R}}(\hat{\varsigma})$ we add a vertex representing the address read and add an edge $\hat{r}_{\hat{\varsigma}} \to \hat{m}_{\hat{\varsigma}}$ to the graph.
    * For every address $\hat{w}_{\hat{\varsigma}}$ in $\hat{\mathcal{W}}(\hat{\varsigma})$ we add a vertex representing the address written and add an edge $\hat{m}_{\hat{\varsigma}} \to \hat{r}_{\hat{\varsigma}}$ to the graph.

Example 5.12. (Abstract resource dependence graph) We continue Example 5.10 on page 36, which demonstrated how some (context-insensitive) marks are attached to three continuations needed during abstract interpretation. We assume that $\hat{\kappa}_0$ was marked with `cons`. Also, $\hat{\kappa}_{20}$ was marked with λ_{12} and $\hat{\kappa}_{24}$ with λ_{16}. Suppose state exploration has terminated. We must now walk the stack of each state and aggregate marks. We look at three evaluation states in which our marks are involved.

- In the $\widehat{EvalTail}$ state $\hat{\varsigma}_1$ for (`set! z 123`) (body of `writez`) with stack $\hat{\kappa}_{20} \to \hat{\kappa}_0$, marks λ_{12} and `cons` are collected. We have $\hat{\mathcal{R}}(\hat{\varsigma}_1) = \emptyset$ and $\hat{\mathcal{W}}(\hat{\varsigma}_1) = \{\texttt{z}\}$. Hence, both these procedures will have a write dependency on resource `z`.
- In the $\widehat{EvalTail}$ state $\hat{\varsigma}_2$ for `z` (body of `readz`) with stack $\hat{\kappa}_{24} \to \hat{\kappa}_0$, marks λ_{16} and `cons` are collected. Here $\hat{\mathcal{R}}(\hat{\varsigma}_2) = \{\texttt{z}\}$ and $\hat{\mathcal{W}}(\hat{\varsigma}_2) = \emptyset$. Therefore, both these procedures will have a read dependency on resource `z`.
- Finally, in the $\widehat{EvalTail}$ state $\hat{\varsigma}_3$ for (`cons p1 p2`) with stack $\hat{\kappa}_0$, mark `cons` is collected and given a read dependency on resources `p1` and `p2` because $\hat{\mathcal{R}}(\hat{\varsigma}_3) = \{\texttt{p1}, \texttt{p2}\}$ and $\hat{\mathcal{W}}(\hat{\varsigma}_3) = \emptyset$. As a result, the expression (`cons p1 p2`) has both a write and read dependency on resource `z`.

The complete resulting abstract resource dependence graph was given in Figure 5.1 on page 28, and it includes the dependencies we resolved in Example 5.12 above.

5.4.9 Computing value flow

Besides an abstract resource dependency graph, IPDA can also compute a monovariant aggregate store that maps identifiers onto the possible values they represent during program execution. Closures are given special treatment and are stripped of their binding environment, leaving only the lambda expression. Possibly the most useful information we can obtain through the aggregate store is that for every identifier in operator position we can determine the set of procedures that flow to a specific application site.

To compute the monovariant aggregate store, we first traverse all explored states and join together all state stores to obtain the (possibly polyvariant) aggregate store

$$\hat{\sigma}_{\hat{\mathcal{V}}} = \bigsqcup_{\hat{\varsigma} \in \hat{\mathcal{V}}} \hat{\sigma}_{\hat{\varsigma}}$$

Joining stores is conceptually simple. If $\hat{\sigma}_1 \sqcup \hat{\sigma}_2 \sqcup \cdots \sqcup \hat{\sigma}_n = \hat{\sigma}$, then for all addresses $\hat{a}$: $\hat{\sigma}(\hat{a}) = \hat{\sigma}_1(\hat{a}) \sqcup \hat{\sigma}_2(\hat{a}) \sqcup \cdots \sqcup \hat{\sigma}_n(\hat{a})$.

The monovariant aggregate store $\mu : \mathsf{Var} \to \widehat{Val}^*$ is a store that maps identifiers to values. We populate it by selecting all bindings from the aggregate store $\hat{\sigma}_{\hat{\mathcal{V}}}$ by stripping all context information from bindings, yielding variables. We then aggregate all possible values per variable.

5.5 Optimizing the implementation of the analysis

If the analysis would be implemented exactly as it was presented in previous sections, it would be quite inefficient. There are however some clever optimizations we can apply to mitigate much of the memory and space consumption required by naive implementations.

Without going into formal details, we describe two techniques. The reader may want to consult the cited references for a more in-depth treatment.

5.5.1 Widening

Keeping a list of states and checking if a state already exists for every newly created state is very inefficient. In order to improve the speed of the analysis, we can implement *per-context widening* [18]. It is based on the fact that expressions, binding environment, return points and continuations, etc. can be efficiently compared using equality. This is not necessarily true for stores, which may contain many entries and monotonically grow in size. If we use only one store per context, we can greatly speed up analysis time.

For example, the context of the $\widehat{EvalTail}$ state is $(e, \hat{\beta}, \hat{rp}, \hat{t})$, i.e. everything but the store. Similarly, we can define contexts for the three other states. Using a hash table we can then map every context onto a single store, instead of maintaining a big list of states with each state containing its proper store. To obtain the state store, we now have to look it up using the state context.

5.5.2 Abstract garbage collection

It is in the nature of our abstract machine to frequently re-use abstract addresses for allocation, even though there are free addresses available. As a consequence multiple abstract values are forced to reside at the same slot in the store. By performing *abstract garbage collection* we can limit this kind of merging, thereby increasing the precision of the analysis [20].

Abstract garbage collection can also be called narrowing, since it is the opposite of widening. Abstract garbage collection is similar to its concrete counterpart. For a certain state we can calculate the addresses that are referenced by that state, called the root addresses. Using those roots we can transitively determine which addresses are reachable. Addresses that are not reachable can be removed from the store. The result is a store that is more precise than before garbage collection occurred.

Abstract garbage collection is important to increase the precision of the analysis. Abstract garbage collection is also an interesting case to illustrate how abstractions must remain sound: if we perform abstract garbage collection, we *must* perform concrete garbage collection as well, otherwise there would exist concrete values that have no abstract counterpart. This violates the guarantee offered by our abstractions that every possible concrete value that may exist has an abstract counterpart.

5.5.3 Abstract Counting

Abstract counting makes use of the simple observation that an abstract address that is allocated *exactly once* can only have one concrete counterpart. So while abstract addresses can represent multiple concrete addresses, using abstract counting we can determine when they correspond to their concrete counterpart.

Abstract counting is done at the level of the store. The store keeps track of the number of allocations of addresses, assigning each address a cardinality. In order to provide the right precision to implement abstract counting, the cardinality is an element of the set $\{0, 1, \infty\}$.

Abstract counting is useful when we update the value at a certain abstract address. If the cardinality of that address is 1, then it is sound to consider any update to be a *strong update*, effectively replacing the previous value with the new one. If, on the other hand, the cardinality is ∞, then we have to fall back to a *weak update* or a join, because the abstract address possibly represents multiple concrete addresses. This diminishes the precision of the analysis.

Equal abstract addresses with cardinality 1 also implies equality for the corresponding concrete addresses, meaning that they *must* alias. From this point of view, abstract counting can increase the power of an analysis.

Note that the verb “counting” here only applies to the number of allocations of addresses, *not* to the number of concrete values represented by the abstract value that an abstract address points to.

The number of distinct abstract addresses used for a variable depends on a combination of factors.

- The underlying program of course determines where and when variables get bound. It is clear that binding variables inside a loop can only increase the chance that abstract addresses will be reused, their cardinality becoming ∞.

Figure 5.10 Abstract counting: Single binding of variable `z`.

```
(let*
  ((z 0)
   (writez (lambda () (set! z (+ z 1)))))
  (writez)
  z)
```

Figure 5.11 Abstract counting: multiple bindings of variable `n`.

```
(letrec
  ((f (lambda (n)
        (if (zero? n)
            0
            (f %36% (- n 1))))))
  (f %45% 100))
```

- The precision of the primitive operations and special forms like `if` can influence the times that the abstract interpreter enters blocks of code that bind variables, and therefore also indirectly influences the chances of rebinding an existing abstract address.
- The sensitivity of the value-flow analysis (k) influences the polyvariance of the abstract addresses. The higher k is, the more opportunities there are for creating fresh abstract addresses (with cardinality 1) during interpretation.
- Abstract garbage collection can be considered as an operation that sets the cardinality of unused addresses to 0. Although abstract garbage collection is orthogonal to abstract counting, it increases the usefulness of the latter.

Example 5.13. In the program depicted in Figure 5.10, value-flow analysis would compute the set of possible answers of this program as $\{0, 1\}$. It is clear however that the address for `z` is only bound once during execution. Therefore the analysis can decide that the update of `z` through `set!` is in fact a strong update, overwriting any previous value in the store. Hence the computed result of this program will be simply $\{1\}$, making the analysis more precise when abstract counting is applied.

Example 5.14. The program in Figure 5.11 implements a countdown loop. Suppose that $k = 1$ and the precision of primitive operations `zero?` and $-$ is sufficiently high (i.e. the return value is an abstract value representing one concrete value during the following discussion). We can trace the successive bindings for formal parameter `n` as the interpreter enters the body of `f` follows. After applying `f` at call site 45 there will be only one allocation of binding $(\mathtt{n}, 45)$ with value $\langle 100 \rangle$ so its cardinality is and remains 1. After applying `f` at call site 36 a first time, $(\mathtt{n}, 36)$ is allocated with cardinality 1 value $\langle 99 \rangle$. The interpreter at least evaluates call site 36 once more and binding $(\mathtt{n}, 36)$ is reallocated, but this time its cardinality becomes ∞ and its value is weakly updated to $\langle 99, 98 \rangle$.

5.6 Conclusion

We presented the analysis by Might and Prabhu to approximate dependencies that may arise when expressions invoke procedures in a higher-order program [19]. The analysis is developed using abstract interpretation as a framework. It uses an abstract state machine that allows inspection of stack configurations per state. By determining the addresses read and written per state and applying Harrison's principle, we can construct an abstract resource dependency graph relating addresses and procedure invocations. This information, together with the dependency analysis from Chapter 2 allows us to conservatively determine all dependencies between all expressions in a program.

Chapter 6

Implementation

6.1 Introduction

All the ideas and techniques presented in this dissertation are implemented in a Java project called Streme. The name Streme refers to Scheme (the language) and "stream" (because it uses a queue-based work-stealing algorithm in its evaluator).

The code can be found at `http://code.google.com/p/streme`.

6.2 Parallel constructs

Streme implements an evaluator that understands a subset of R5RS Scheme with several additional parallel constructs. The basic construct for expressing parallelism is the special form `future` and its companion procedure `touch`.

- `(future e)` allows the evaluator to evaluate expression `e` immediately or at some unspecified point in the future, and possibly in parallel with the rest of the program. It may return a *future value* (also called a *promise*) representing the future computation, or it may also return the value of `e`.
- `(touch f)` awaits the termination of the future computation represented by `f` and returns that computation's value, or if `f` is not a future value then it returns the value of `f`.

From these descriptions it is clear that `future` and `touch` have *intentional semantics* [9] in Streme. This means that `future` is not simply `fork` or `spawn` for example, where evaluation is unconditionally started in another thread and `touch` plays the role of a `join` that returns a value. `future/touch` is not `delay/force` either, computing the value of an expression at the point where it is needed. And both `future` and `touch` are not the identity function `id`, the function that immediately returns its evaluated operand. In fact, `future` and `touch` can be all three of the previous, at the discretion of the evaluator. We will refer to these semantics as *fork*, *delay* and *id semantics* respectively.

In practice, Streme will only apply fork or id semantics. It is clear that an evaluator providing `future/touch` will strive for parallelism using fork semantics wherever it decides it is appropriate. Inappropriate situations could include those in which there are already too many futures running (i.e.

Figure 6.1 `fib` function with future and touch.

```
(define (fib n)
  (if (< n 2)
      n
      (let ((f (future (fib (- n 2)))))
        (+ (fib (- n 1)) (touch f)))))
```

soft or hard limits), or when the overhead of executing a future in a separate thread is not deemed worth the overhead (by employing heuristics or profiling). Id semantics then serves as a fallback. Streme currently only imposes a configurable limit on the number of concurrently running future computations, by default based on the number of processing cores.

It is imperative that `future` and `touch` act as true annotations. It should be possible to eliminate all occurrences of `future` and `touch` from a program without changing the behavior of that program. Consider for example an evaluator that always assigns id semantics to `future` and `touch`, thereby evaluating future expressions immediately.

```
(define (id x) x)
(define future id)
(define touch id)
```

Because the above definitions are sound it becomes clear that removing `future` and `touch` should never have any observable impact. In the presence of side-effects, we can only guarantee this if (the addition of) future expressions correctly takes dependencies into account (§2.2).

Example 6.1. (Future/touch) Figure 6.1 shows a definition of the Fibonacci function using `future` and `touch`. In this particular example it is important that the call `(fib (- n 1))` appears before `(touch f)`. If not, we could start a future computation for which we immediately wait for the result to be returned, yielding sequential evaluation.

`future` and `touch` can be used to define higher-level parallel syntactic constructs. This is the case for Streme's parallel `let` form `let||`. For example, the expression

```
(let|| ((a e1)
        (b e2))
  ebody)
```

is rewritten into

```
(let ((fa (future e1))
      (fb (future e2)))
  (let ((a (touch fa))
        (b (touch fb)))
    ebody)
```

6.3 Architecture

The architecture of Streme follows a typical pipeline design, consisting of various analyzers, compilers and transformers that are chained together to form an evaluator. The stages can be subdivided into a front-end, middle-end and back-end. The middle-end defines abstract syntax tree (AST), together with infrastructure to analyze and transform ASTs. This is where static analyzers and parallel rewriters can be plugged in to do the actual parallelization of programs.

Parser

The parser returns Scheme data like numbers, symbols, and lists that represents the input program. The fact that we can use Scheme data to represent Scheme programs is called homoiconicity, and it remains true even if the underlying implementation language is Java. The parser also expands some syntactic sugar such as apostrophes to a quote form: `'abc` becomes `(quote abc)`.

MacroExpander

The macro expander rewrites derived syntax like `let||`, `or`, `cond`, `case`, `do`, etc. into more essential syntax. The main advantage is that it simplifies further stages of the compiler. They do not need to know how to handle all the derived forms, but only a core syntax. For example, the expression

```
(case x
  ((a b) (quote number))
  ((c) (quote pair))
  (else (quote help)))
```

will be first rewritten by expanding `case` to a `cond` as follows

```
(let ((_t0 x))
  (cond ((memv _t0 (quote (a b))) (quote number))
        ((memv _t0 (quote (c))) (quote pair))
        (else (quote help))))
```

Then the `cond` is expanded into an `if` expression

```
(let ((_t0 x))
  (if (memv _t0 (quote (a b)))
      (quote number)
      (if (memv _t0 (quote (c)))
        (quote pair)
        (quote help))))
```

Finally, the `let` expressions can be desugared into `lambda` expressions and function application if desired. However, in our approach for parallelization we want to keep `let` expressions intact. There is also a flag for keeping `letrec`s from being rewritten into more basic syntax. The macro-expander expands into the following syntax:`if`, `begin`, `define`, `future`, `lambda`, `quoted`, `set!`, and optionally `let` and `letrec`.

Data2AstCompiler

This compiler takes Scheme data and transforms it into an abstract syntax tree (AST). The following node types are defined in the AST:

- Application
- Begin
- Binding (used in different `let` forms)
- Define
- Future
- If
- Lambda
- Let (used for `let`, `let*`, `letrec`, `let||`)
- Quoted
- Ref
- SetVar
- Var

AlphaConverter

Alpha conversion introduces unique identifiers for every variable name in a program. This makes it easier to reason about code and introduce changes. Alpha conversion uses a renaming strategy that is also used by other stages that need to introduce unique names.

AnfConverter

The AnfConverter converts an AST into a specific form described in Figure 3.2 on page 20.

IpdaAnalyzer

Interprocedural dependency analyzer, implemented as an abstract interpreter that takes ANF input and adds several elements like bindings, binding environments, return points, closures, stores, continuations and marks. It builds the abstract resource dependency graph (§5.4.8).

It implements all of the tricks that we discussed in §5.5: it features per-context widening, abstract counting and abstract garbage collection.

BruteForceParallelizer

Implements the approach outlined in Chapter 2. Takes the results from the IpdaAnalyzer for interprocedural dependencies, and adds logic for dealing with all other types of dependencies.

Simplifier

Performs some "peephole" optimizations on the level of the AST. For example, `(begin e)` is substituted by the single expression `e` just as `(let ((u e)) u)` is replaced by `e`

CoreSchemeRewriter

The CoreSchemeRewriter is the equivalent of the MacroExpander, but on the level of the AST. It rewrites the AST to CoreScheme, which comprises the following syntax: `if`, `begin`, `define`, `future`, `lambda`, `quoted`, `set!`.

Ast2ExpCompiler

Compiles CoreScheme AST into expressions that know how to evaluate themselves [3]. Self-evaluating expressions are essentially Java objects that define an evaluate method. The evaluate method is called passing an evaluation context that provides access to information and services needed during evaluation.

There are currently two types of expression classes, each with its own compiler:

1. EXP-S: uses the JVM stack for evaluation.
2. EXP-C: continuation-passing evaluator.

EXP-S

Expression classes that use the JVM stack for evaluation and procedure application. The signature of the evaluate and apply methods are

```
class Exp
{
  Object eval(StremeContext context, Env env);
}

class Procedure
{
  Object applyN(StremeContext context, Object[] operands, Env env);
}
```

and the expression `ExpApp2`, which applies two operands to an operator, is implemented as

```
class ExpApp2 extends Exp
{
  private Exp operator;
  private Exp operand1;
  private Exp operand2;

  public Object eval(final StremeContext context, final Env env)
  {
```

```
    Procedure eoperator = (Procedure) operator.eval(context, env);
    Object eoperand1 = operand1.eval(context, env);
    Object eoperand2 = operand2.eval(context, env);
    return eoperator.apply2(context, eoperand1, eoperand2, env);
  }
}
```

In `future` expressions, implemented by `ExpFut`, the context is accessed to submit a task representing the evaluation of the future expression. This task can be executed asynchronously. Tasks are submitted as `Runnable`s

```
public interface Runnable
{
  void run();
}
```

The maximum number of futures submitted in the implementation below is limited to $n - 1$, where n is the number of threads available as reported by the context.

```
public class ExpFut extends Exp
{
  private static AtomicInteger futures = new AtomicInteger(0);
  private Exp value;

  public Object eval(final StremeContext context, final Env env)
  {
    if (numberOfFutures.get() > context.numberOfThreads() - 2)
    {
      // evaluate immediately and return result return
      value.eval(context, env);
    }
    else // room for a future
    {
      numberOfFutures.incrementAndGet();
      final Future future = new Future();
      context.execute(new Runnable() // submit for asynchronous execution
      {
        // asynchronously called
        public void run()
        {
          future.set(value.eval(context, env));
          numberOfFutures.decrementAndGet();
        }
      });
      return future;
    }
  }
}
```

A future itself is simply an object allowing for a synchronized hand-off of a value (the evaluated future expression).

```
public class Future
{
  private boolean done;
  private Object value;

  public synchronized Object get() throws InterruptedException
  {
    while (!done)
    {
      wait();
    }
    return value;
  }

  public synchronized void set(Object value)
  {
    this.value = value;
    done = true; notifyAll();
  }
}
```

The `touch` procedure checks if its argument is a future. If so, it will try to retrieve the value from the future, blocking until it becomes available. To implement intentional semantics, `touch` will simply return any non-future argument it receives as its value.

```
class Touch extends Procedure
{
  public Object apply1(StremeContext context, Object operand, Env env)
  {
    if (operand instanceof Future)
    {
      Future future = (Future) operand;
      return future.get(); // blocking call
    }
    else // operand not a future value
    {
      return operand;
    }
  }
}
```

This evaluator is not properly tail-recursive, and it has no support for implementing (and does not implement) `call-with-current-continuation`.

EXP-C

A second evaluator that exclusively uses continuations for evaluation. Signatures for the eval and apply method are

```
class Exp
{
  void eval(StremeContext context, Env env, Cont cont);
}

class Procedure
{
  void applyN(StremeContext<Exp> context, Object[] operands, Env env, Cont cont);
}
```

Expressions pass a value to the continuation instead of returning it using the JVM stack.

```
interface Cont
{
  void call(Object val);
}
```

Expression `ExpApp2` in continuation-passing style exhibits the typical waterfall pattern by chaining continuations and becomes

```
public class ExpApp2 extends Exp
{
  private Exp operator;
  private Exp operand1;
  private Exp operand2;

  public void eval(final StremeContext context, final Env env, final Cont cont)
  {
    operator.eval(context, env, new RunnableCont(context)
    {
      protected void executeContinuation(final Object eoperator)
      {
        operand1.eval(context, env, new RunnableCont(context)
        {
          protected void executeContinuation(final Object eoperand1)
          {
            operand2.eval(context, env, new RunnableCont(context)
            {
              protected void executeContinuation(Object eoperand2)
              {
                ((Procedure) eoperator).apply2(context, eoperand1, eoperand2, env, cont);
              }
            });
```

```
          }
        });
      }
    });
  }
}
```

A `RunnableCont` is an implementation of a continuation that uses the context to submit itself as task.

```
public abstract class RunnableCont extends Cont implements Runnable
{
  private StremeContext context;
  private Object value;

  // asynchronously called
  public final void run()
  {
    executeContinuation(value);
  }

  // implements the rest of the computation this continuation represents
  protected abstract void executeContinuation(Object value);

  public void call(Object value)
  {
    this.value = value; // the value with which this continuation is called
    context.execute(this); // submit for asynchronous execution
  }
}
```

In fact, in the EXP-C evaluator every continuation acts as a future.

Work-stealing

For both evaluators a context must be implemented that provides the possibility of asynchronously executing tasks, submitted as `Runnable`s.

```
public interface StremeContext<T>
{
  GlobalEnv globalEnv();
  Object read(Reader reader);
  T compile(Object source);
  void execute(Runnable runnable);
  int numberOfThreads();
}
```

For this purpose a `ThreadPoolExecutor` is used, implementing a work-stealing strategy to realize parallelism in Streme. Tasks (futures and/or continuations) are submitted onto a work queue. A

pooled worker thread that is idle "steals" a task from the work queue and executes it. During execution, the task can put one or more tasks back onto the queue.

6.4 Conclusion

We described some details of our Scheme interpreter, called Streme. The front- and middle-end of Streme has a modular architecture and consists of a pipeline of program analysis and transformation tools. For the backend two evaluators were implemented: one uses the JVM stack during evaluation and the other is written in continuation-passing style. Both evaluators implement the basic parallelization constructs `future` and `touch` with intentional semantics.

Streme was developed so it could be used for our experiments (Chapter 7).

Chapter 7

Experiments and validation

7.1 Introduction

In order to validate the static analysis we carried out experiments by parallelizing and running several Scheme benchmarks. The experiments were performed with the EXP-S evaluator using the JVM stack as backend. The fact that continuations as tasks are far more fine-grained than futures caused unpredictable overhead in the EXP-C evaluator and most of the benefits gained from parallel execution were lost.

Every benchmark used $k = 1$ as setting for the interprocedural analysis.

The experiments were carried out on a Sun Java 1.6.0 Runtime Environment with HotSpot 64-bit Server JVM running an Intel machine with 8 Xeon processors. Every benchmark was run with 1 up to 16 threads, the number of cores reported by the Java runtime.

Running the benchmarks without a properly tail-recursive Scheme implementation required a bigger JVM stack size than the default. One experiment required 64 MB of stack size (per thread). Therefore, the JVM was started with the following arguments: `-Xmx1024M -Xss64M`. All the benchmarks run and terminate properly with increased stack size, yielding correct values. To verify correctness, we used Racket in R5RS mode [23].

In the performance graphs we plot out the running time of the automatically *parallelized* benchmark (in ANF style) against the number of threads. In principle we should make a comparison with the running time of the *sequential* ANF version of the benchmark. The difference however between the running time of the sequential ANF version versus that of the parallelized benchmark running on one thread was small for all benchmarks. This is because the Streme evaluator falls back to id semantics for both future and touch. Nevertheless the sequential ANF timings are included in the global overview presented in Table 7.1 on page 65.

7.2 Benchmark experiments

7.2.1 `fib`

Description The function for calculating the n-th Fibonacci number is one of the best known functions and benchmarks. We have already encountered it before in in ANF style. For the benchmark we

Figure 7.1 Computational tree for Fibonacci function

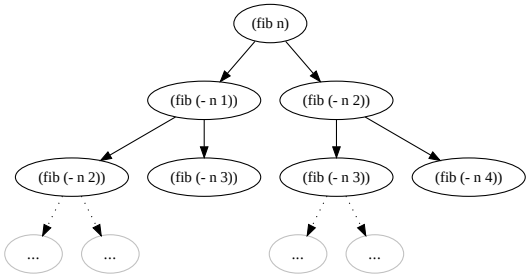
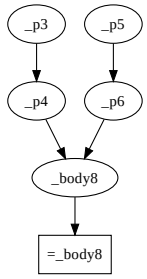
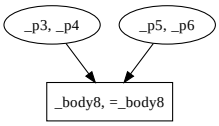


Figure 7.2 `fib` function in ANF style, with binding dependency graph and binding order graph

```
(letrec ((_fib0
          (lambda (_n1)
           (let ((_p2 (< _n1 2)))
            (if _p2
                _n1
                (let ((_p3 (- _n1 2)))
                 (let ((_p4 (_fib0 _p3)))
                  (let ((_p5 (- _n1 1)))
                   (let ((_p6 (_fib0 _p5)))
                    (+ _p4 _p6)))))))))))
 (_fib0 40))
```





use the "standard" direct-style tree-recursive version of the function with `(< n 2)` as condition for termination, and we perform 4 iterations.

```
(letrec  ((fib (lambda (n)
                 (if (< n 2)
                     n
                     (+ (fib (- n 2)) (fib (- n 1)))))))
  (fib 40))
```

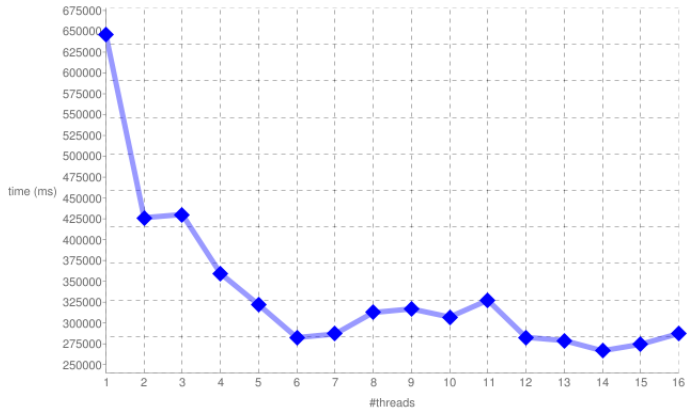
If we look at the tree of computations generated by the `fib` function (Figure 7.1), in which every non-leaf node depends on its children, we can deduce two things [31]:

1. Considering the number of times the same expression is evaluated, the double-recursive version of `fib` is not the most efficient way of calculating Fibonacci numbers.

2. There is always more work on the left side of the tree, although larger values for `n` makes this difference relatively smaller higher up in the tree, resulting in only a slight imbalance. This should be a favorable condition for successful parallel evaluation.

Dependency analysis Figure 7.2 shows the actual ANF version of the `fib` function that is generated and used during analysis, together with the binding dependency graph and binding order graph for the nested `let`s. The opportunity for parallelization is clearly present in the graphs (§2.2).

Figure 7.3 Performance of `fib` benchmark, 4 iterations



Parallelization The automatically parallelized version of the `fib` function is as follows.

```
(letrec ((_fib0
          (lambda (_n1)
           (let ((_p2 (< _n1 2)))
            (if _p2
                _n1
               (let ((_p3 undefined)
                     (_p5 undefined))
                (let|| ((_p4 (begin (set! _p3 (- _n1 2)) (_fib0 _p3)))
                        (_p6 (begin (set! _p5 (- _n1 1)) (_fib0 _p5))))
                 (+ _p4 _p6)))))))))
 (_fib0 40))
```

It contains no surprises. There is one `let||` that calculates children `(fib (- n 1))` and `(fib (- n 2))` to obtain a value for `(fib n)`.

Performance The performance of the `fib` benchmark is as could be expected. The biggest gain in performance comes when going from one to two evaluation threads. After that, the execution time still generally drops with the addition of threads, until the graph flattens out at six threads.

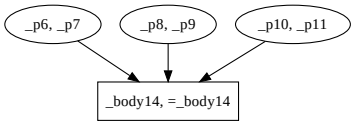
7.2.2 `tak`

Description The `tak` function, named after Ikuo Takeuchi, is also a well-known Gabriel benchmark involving recursion. The version we use is not the original one, but the version of McCarthy returning `z` when `(not (< y x))` [32]. We do 4 iterations of this benchmark.

```
(letrec ((tak (lambda (x y z)
                (if (not (< y x))
                    z
                    (tak (tak (- x 1) y z)
                         (tak (- y 1) z x)
```

Figure 7.4 `tak` function in ANF style, with binding dependency graph and binding order graph

```
(letrec
 ((_tak0
   (lambda (_x1 _y2 _z3)
    (let ((_p4 (< _y2 _x1)))
     (let ((_p5 (not _p4)))
      (if _p5
          _z3
          (let ((_p6 (- _x1 1)))
           (let ((_p7 (_tak0 _p6 _y2 _z3)))
            (let ((_p8 (- _y2 1)))
             (let ((_p9 (_tak0 _p8 _z3 _x1)))
              (let ((_p10 (- _z3 1)))
               (let ((_p11 (_tak0 _p10 _x1 _y2)))
                (_tak0 _p7 _p9 _p11)))))))))))))
  (_tak0 40 30 20))
```




```
                          (tak (- z 1) x y))))))
  (tak 40 30 20))
```

Dependency analysis Figure 7.4 shows the ANF version of the `tak` function. Also shown are the binding graphs for the nested `let`s inside the `if` expression, which constitute the only parallelization opportunity.

Parallelization After dependency analysis and program transformation, the result is a parallel `tak` function containing a `let||` with three branches.

```
(letrec ((_tak0
          (lambda (_x1 _y2 _z3)
           (let ((_p4 (< _y2 _x1)))
            (let ((_p5 (not _p4)))
             (if _p5
                 _z3
                 (let ((_p6 undefined) (_p8 undefined) (_p10 undefined))
                  (let|| ((_p7 (begin (set! _p6 (- _x1 1)) (_tak0 _p6 _y2 _z3)))
                          (_p9 (begin (set! _p8 (- _y2 1)) (_tak0 _p8 _z3 _x1)))
                          (_p11 (begin (set! _p10 (- _z3 1)) (_tak0 _p10 _x1 _y2))))
                   (_tak0 _p7 _p9 _p11)))))))))
 (_tak0 40 30 20))
```

Performance The performance graph of `tak` bears resemblance to the performance graph of `fib`. However, `fib` easily breaks the -50% evaluation time barrier at 5 threads, while `tak` scratches -50% at around 10 threads. Still, the `tak` benchmark confirms that `fib` is not a fluke, and that high levels of recursion can offer good results with our method of automatic parallelization.

Figure 7.5 Performance of `tak` benchmark, 4 iterations

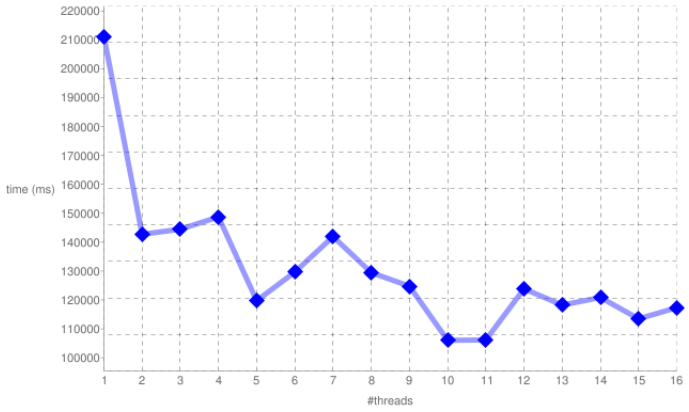
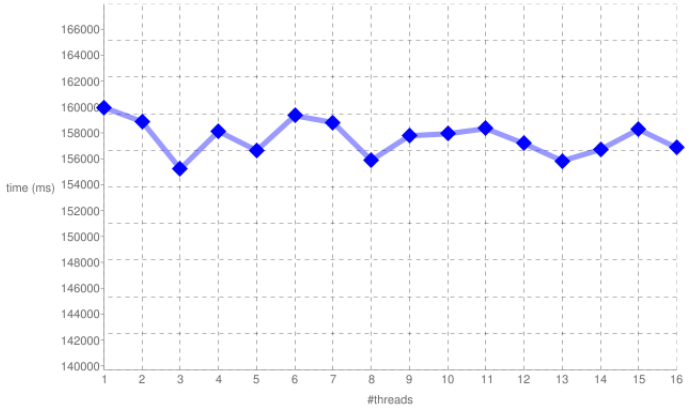


Figure 7.6 Binding order graph for `nboyer`

Figure 7.7 Performance of `nboyer` benchmark, 4 iterations



7.2.3 `nboyer`

Description `nboyer` is an updated version of the `boyer` Gabriel benchmark, and principally tests lists, vectors and garbage collection [2, 25]. It is representative of real AI applications because it performs Prolog-like rule-directed rewriting. As such it is a logic programming benchmark and in comparison to `fib` and `tak` certainly not a micro-benchmark. We use `n=3`, resulting in 5375678 rewrites occurring, and we repeat the benchmark 4 times. The program is too big to be included here – we refer the reader to [25].

Dependency analysis The binding order graph for `nboyer` (Figure 7.6) is not very different from its binding dependency graph.

Performance The performance graph for `nboyer` (Figure 7.7) is quite unexciting and the only sound conclusion we can draw is that brute-force parallelization has no significant effect on the running time of this benchmark. Clearly, our parallelization approach is not suited for every type of program. The positive news is that in case we do not have a speedup, our automatic parallelization method does not necessarily cripple performance either, as this benchmark illustrates.

7.2.4 `quicksort`

Description `quicksort` is an efficient sorting algorithm developed by C. A. R. Hoare in 1960. We use a straightforward implementation shown below [17] to sort a *fixed* list `*random-list-40000*` of 40000 generated pseudo-random integers in the range $0 \ldots (2^{17} - 1)$. For the benchmark timing we measure how long it takes to sort this list 10 times.

```
(define pivot (lambda (l)
              (cond ((null? l) 'done)
                  ((null? (cdr l)) 'done)
                    ((<= (car l) (cadr l)) (pivot (cdr l)))
                  (else (car l)))))

(define partition (lambda (piv l p1 p2)
                    (if (null? l)
                        (list p1 p2)
                          (if (< (car l) piv)
                            (partition piv (cdr l) (cons (car l) p1) p2)
                           (partition piv (cdr l) p1 (cons (car l) p2))))))

(define (quicksort l)
  (let ((piv (pivot l)))
    (if (equal? piv 'done)
        l
        (let ((parts (partition piv l '() '())))
          (append (quicksort (car parts))
                  (quicksort (cadr parts)))))))

(quicksort *random-list-40000*)
```

Dependency analysis After dependency analysis, no less than four opportunities for parallelization are discovered. Three of them are trivial however, including only applications of `car`, `cdr`, and `cons`, like for example (after ANF conversion)

```
(let|| ((_p21 (cdr _l6))
        (_p23 (begin (set! _p22 (car _l6)) (cons _p22 _p17))))
  ...)
```

The most promising parallelization is that of the recursive calls to `quicksort`.

Figure 7.8 Performance of `quicksort` benchmark after brute-force automatic parallelization, 10 iterations

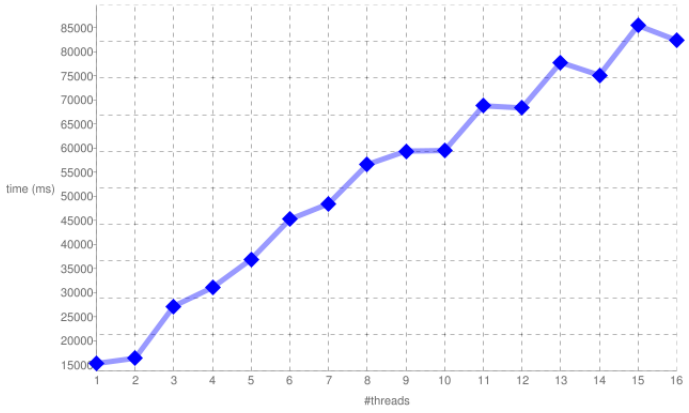
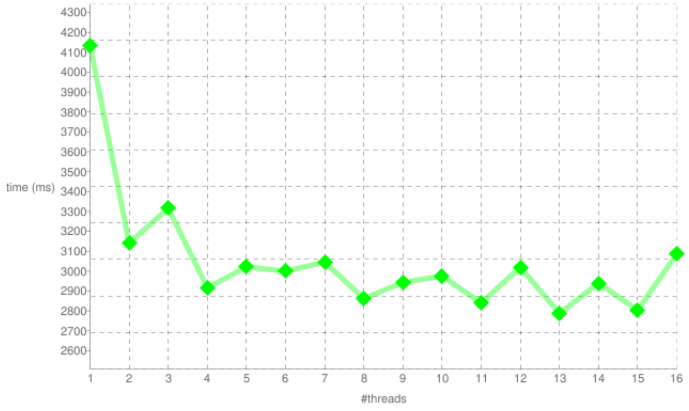


Figure 7.9 Performance of *manually parallelized* `quicksort` benchmark, 10 iterations

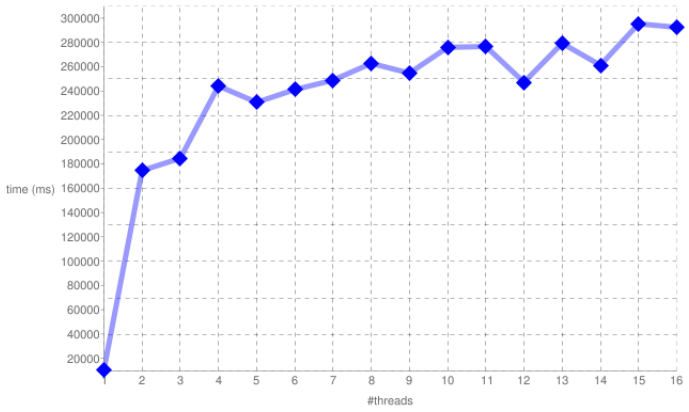



```
(let|| ((_p29 (begin (set! _p28 (car _parts10)) (_quicksort0 _p28)))
        (_p31 (begin (set! _p30 (cadr _parts10)) (_quicksort0 _p30))))
  ...)
```

Performance Clearly the brute-force automatic parallelization has a detrimental effect on the running time (Figure 7.8), even though `quicksort` employs a divide-and-conquer strategy which should make it a good candidate for receiving a boost in speed when running on multiple cores. *Manual parallelization* of the input program by (only) parallelizing the recursive calls to `quicksort` yields the performance graph in Figure 7.9, showing that the runtime of Streme is not the bottleneck.

We can draw the following conclusion from this benchmark: without heuristics guiding the parallelization (certainly) and without more aggressive optimization of the ANF code (possibly) any gain in performance can be offset by overhead introduced by the code transformations as a result of brute-force parallelization.

Figure 7.10 Performance of `nqueens` benchmark, 10 iterations



7.2.5 nqueens

Description The n-queens problem is a combinatorial problem in which it is required to place n queens on an $n \times n$ chessboard so that no two queens attack each other. In our test suite, we take $n = 10$ and we perform 10 iterations of the benchmark. The code for `nqueens` is shown below.

```
(define (nqueens n)
  (define (_1-to n)
    (let loop ((i n) (l '()))
      (if (= i 0) l (loop (- i 1) (cons i l)))))
  (define (my-try x y z)
    (if (null? x)
      (if (null? y) 1 0)
      (+ (if (ok? (car x) 1 z)
           (my-try (append (cdr x) y) '() (cons (car x) z))
           0)
         (my-try (cdr x) (cons (car x) y) z))))
  (define (ok? row dist placed)
    (if (null? placed)
      #t
      (and (not (= (car placed) (+ row dist)))
           (not (= (car placed) (- row dist)))
           (ok? row (+ dist 1) (cdr placed)))))
  (my-try (_1-to n) '() '()))

(nqueens 10)
```

Dependency analysis After analysis, three opportunities for parallelization are discovered. Similarly to `quicksort` however, the work performed in the branches of the three `let||` is minimal.

name	iter	analysis	f/t	ast	anf	×1	×2	×4	×8
fib	4	101	2/2	174313	666681	645807	-33.99%	-44.35%	-51.6%
tak	4	170	3/3	68466	239765	211236	-32.44%	-29.58%	-38.71%
nboyer	4	10104	87/87	61987	180108	159960	-0.68%	-1.15%	-2.55%
quicksort	10	20016	10/10	4052	14220	15286	+7.22%	+103.17%	+270.09%
nqueens	10	94	6/6	2792	11296	11088	+1651%	+1991%	+2230%

Table 7.1: Overview of all the benchmarks

Performance The performance degradation of the automatically parallelized version of `nqueens` shows worse degradation than that of `quicksort`. In this case, there is no straightforward manual parallelization possible, so contrary to `quicksort` simple heuristics will not solve the problem. Although our brute-force approach to parallelization does not necessarily cripple the performance of parallel execution (see `nboyer`), it potentially does.

7.2.6 Overview

Table 7.1 presents an overview of all the benchmarks that were ran, together with some key data. Below is a description of each column in the table.

name — name of the benchmark

iter — number of iterations

performance — performance graph of the runtime of the automatically parallelized program on one up to sixteen threads

analysis — analysis time (dependency analysis + parallelization), in ms

`f/t` — number of `future/touch` in parallel ANF

ast — running time of original input program (before transformation), in ms

anf — running time of input program after ANF conversion, in ms

×1 — running time of automatically parallelized program on one thread, in ms

×2 — percentual speedup of automatically parallelized program running on two threads versus running on one thread

×4 — percentual speedup of automatically parallelized program running on four threads versus running on one thread

×8 — percentual speedup of automatically parallelized program running on eight threads versus running on one thread

7.3 Conclusion

It is difficult to draw firm conclusions after running only a limited number of benchmarks. There are however several observations we can make.

- A parallel speedup is possible when double recursion or even higher levels of recursion are present (for example `fib`, `tak`, manual `quicksort`).
- A parallel execution speedup is not guaranteed when double recursion or higher is present (e.g. brute-force `quicksort`). There are too many factors that can influence the outcome, and can do so in a negative way. The absence of heuristics may result in parallelizing trivial pieces of code, negating the possible performance gains that can be realized.
- Our brute-force method is not generally applicable. More general benchmarks and programs will not benefit from it. This does not come as a surprise however, although it has proved to be more difficult than expected to find "real-life" benchmarks that get a boost in performance after automatic parallelization.

Further analysis of the results is needed to determine where improvements can be made. This has not really been done (§8.2) but, given the circumstances in which the experiments were conducted, may prove to be difficult.

- We execute experiments on a Java Virtual Machine that manages its own runtime system. It features a JIT compiler, performs garbage collection, etc. all of which may influence the results in unpredictable ways.
- The underlying hardware also uses various tricks – for example throttling CPU frequencies – and it is difficult to assess how this can affect the results.
- We use a `java.util.ThreadPoolExecutor` to implement work-stealing. It is however a general-purpose implementation of a thread pool with no real support for advanced work-stealing strategies or monitoring thereof. In order to detect performance bottlenecks in this setup, we must be able to pinpoint and quantify parallel idleness and overhead [31].
- We did not take a close look at how the interprocedural analysis (Chapter 5) influenced the results. For small programs, the analysis can be done on sight, but for non-trivial programs examining the results of the analysis can be a difficult task. Additionally, all benchmarks were more or less "typical" Scheme programs, and interprocedural dependencies were not the determining factor and often completely absent. Of course, proving the absence of this type of dependencies remains crucial in our approach.

Chapter 8

Conclusion

This dissertation defended the thesis that automatic parallelization of Scheme programs using static analysis is feasible and allows certain sequential programs to run faster on multicore systems.

The first part of the thesis, concerning the use of static analysis for automatic parallelization, has certainly been developed in quite some detail (Chapters 2 and 5). All the ideas and techniques were implemented and produced consistent results (Chapter 6), showing that it is indeed possible to use static analysis to parallelize sequential programs.

We only had very limited success in supporting the second part of the thesis, concerning the speedup of automatically parallelized programs (Chapter 7). Few benchmarks, mostly the small ones, actually profited from evaluation in Streme on parallel hardware.

8.1 Related Work

8.1.1 Early compilers

In this section we briefly discuss some of the first Scheme compilers that performed optimizations through analysis.

Rabbit

In 1978 Steele wrote his master dissertation on the Rabbit compiler [30]. It was a Scheme compiler that demonstrated the usefulness of converting to continuation passing style (CPS) as an intermediate representation for analyzing and compiling functional programs.

Optimization in Rabbit was limited and is performed before CPS conversion. Rabbit does constant expression folding, simple `if` rewriting, and implements some rewriting rules that together make up a form of β-conversion. These rules check for side-effects and employ heuristics to decide whether they are applicable. The inverse of β-conversion, the elimination of common subexpressions, is not attempted by the compiler.

Rabbit performs some checks to determine whether lambda expressions are treated as data and therefore must actually be closed over their free variables. The compiler also does some simple escaping analysis to determine whether closures may be stack-allocated or must be heap-allocated.

The treatment of side-effects introduced by `set!`, `set-car!`, `set-cdr!`, `cons` and input/output operations are treated as special cases during analysis. Especially `set!` posed the biggest challenges in Rabbit, and some optimizations are simply disallowed in the presence of it[1].

Steele remarks that when writing his dissertation there was no literature available on the subject of optimizing compilers for higher-order languages, and speculates that this was because most optimizing compilers at the time had been for Algol-like languages which do not handle higher-order functions.

Orbit

Orbit was an optimizing compiler similar in spirit to Rabbit [1]. Just like Rabbit it uses CPS conversion as an intermediate representation, but Orbit attempts to optimize more aggressively. It featured interprocedural escape analysis to identify first-class functions that are guaranteed not to out-live their lexically enclosing environment, enabling functions and their environments to be stack-allocated where possible.

Orbit applies program analysis and simplification, including partial evaluation, entirely on the CPS intermediate form instead of the source code.

Because `set!` in Rabbit was considered awkward during analysis, Orbit featured a technique called *assignment conversion* to make sure that assigned variables always denote values, even in the presence of the general assignment operator `set!`. This way, β-substitution can still be performed freely in program transformations.

Parcel

Parcel was a Scheme compiler that performs interprocedural dependence analysis and program restructuring with the aim to automatically parallelize Scheme programs [11]. Parcel's dependence analysis is accomplished by abstract interpretation that tracks the lifetime of and side-effects upon mutable objects like variables and cons-cells. *Fine-grained* parallelism was already available at the time that Parcel was developed, e.g. in the form of Fortran compilers that where capable of vectorizing loops. In addition, Parcel also extracted *coarse-grained* (or high-level) parallelism from programs, thereby striving to execute lengthy blocks of code concurrently.

Parcel did not use CPS conversion, but was equipped with a complex intermediate form similar to three-address code. Parcel allows variable mutation through `set!` but cons-cells are treated as immutable or are rewritten as closures, which is essentially the opposite of assignment conversion used by Orbit and some of its descendants.

Miprac

Miprac is an interprocedural parallelizing compiler for languages including Scheme, C and Fortran, that addresses some of the shortcomings of Parcel [13, 14]. While Parcel had a more rigid intermediate form specific to Scheme, Miprac operates upon an intermediate form named MIL that is both targetable from a variety of languages, as well as being more easy to analyze and transform.

[1] For example, when Steele mentions that parameter substitution is not attempted in the presence of `set!`, he comments that "`set!` is so seldom used in Scheme programs that this restriction makes very little difference".

MIL was specifically designed with parallelization in mind and it allows a straightforward representation of a variety of language constructs like pointers, type casting, aliasing, and first-class procedures that are difficult to handle by parallelizers.

8.1.2 Control Flow Analysis

0CFA and k-CFA

Independently from Harrison, Shivers developed a method for performing control flow analysis in mostly functional languages like Scheme using abstract interpretation [28]. The idea of Shivers was that the flow analysis should serve as a tool to be used by optimizing compilers, bringing the performance of the code they produce on par with compilers for traditional languages like Fortran and C.

ΔCFA

Shivers' PhD dissertation made the term "control-flow analysis" in the context of higher-order languages almost synonymous with k-CFA. He formally proved the correctness of several of the developed techniques, except for one idea he called "reflow analysis", in which he discussed the need of a more precise treatment of environments during analysis. The problem is that lambda terms do not flow to application sites but closures do, thereby introducing the environment as an extra component to deal with.

Might explored this idea further with Shivers in his PhD dissertation by using *frame strings* in the setting of k-CFA, resulting in ΔCFA [18]. While procedure strings allow reasoning about dependencies by tracking stack changes between the birth and the usage of resources, frame strings enable reasoning about environmental changes between the births of two environments.

8.1.3 Recent parallel Schemes

Schemik

Schemik is a data-parallel dialect of Scheme also inspired by Common LISP [15]. It uses a stack-based evaluator that is a deterministic pushdown automaton consisting of an execution stack for operations and a result stack for data. One of the operations for the execution stack is FEVAL ("fork eval") that will start the evaluation of an expression in an independent evaluator with its own pair of stacks. Schemik is capable of handling side effects introduced by `set!` and primitives like `set-car!`.

Contrary to our approach, Schemik does not employ a brute-force strategy and selects the expressions to parallelize based on the number of transitions it takes to be evaluated.

The `nqueens` benchmark runs faster in parallel using Schemik, although it does not scale as well as the `fib` benchmark.

Racket

Just as Streme, Racket Scheme [23] provides `future` and `touch` as a means to implement parallelism. The notion of "safe" parallelism is different from that described in the context of this dissertation

and different to the implementation in Streme. Safety in Racket is a property inherently tied to the system implementation. Using floating point arithmetic can cause the interpreter to suspend a future computation. As the Racket documentation points out, this and other "unsafe" operations are far from apparent at the level of a Racket program.

8.2 Future work

After examination of the results of the benchmarks, there are many opportunities to improve upon the current approach.

- First and foremost, an effort can be made to make the brute-force approach less brute-force by adding heuristics to guide the parallelization process. This should avoid the kind of situations that we encountered with the `quicksort` benchmark. The idea would be to (partially) answer the question of whether it is *beneficial* to parallelize certain expressions. Parallelizing a single `car` and a `cons` should be avoided for example. Through experimentation it can be deduced which heuristic(s) works best, and in the best case scenario generally applicable heuristics can mend the current approach to yield good results even on `quicksort`.

- We test the relative speedup of the ANF version of programs. When compare the fastest time of the ANF version (on any number of threads), it is still *slower* than running the unmodified input program on one thread. ANF representation therefore is a double-edged sword in our approach. It makes analysis and rewriting more simple, and it should facilitate generating optimized code because it is closer to the machine (by explicitly breaking down expressions into simpler subexpressions), but with the high-level expression evaluators EXP-C and EXP-S the overhead of ANF incurs a performance penalty that can not be undone by parallel evaluation. Therefore we can think of two follow-up scenarios:

  - Either we exploit the fact that ANF code is closer to the machine and generate optimized code from the parallel ANF, yielding more optimal parallel performance.
  - Or we optimize the parallel ANF code more aggressively so that its negative impact on execution time does not eclipse possible benefits of parallelization.

- The bloat from ANF when using high-level evaluators is not the only factor hindering performance. We can generate parallel code from the binding order graph in a more optimized way as well. For example, we assume that every "interior" node from a group of bindings inside a binding node must be visible outside that node. Therefore we create an enclosing `let` to declare the variable, and inside our future expression we `set!` its value (§2.7). This is not always needed. The point we try to make here is the same as the point we made for optimizing the resulting ANF code: optimizing the generated code by further analysis can improve the general results.

Bibliography

[1] N. Adams, D. Kranz, R. Kelsey, J. Rees, P. Hudak, and J. Philbin. ORBIT: an optimizing compiler for scheme. *ACM SIGPLAN Notices*, 21(7):233, 1986.

[2] H.G. Baker. The Boyer benchmark meets linear logic. *ACM Sigplan Lisp Pointers*, 6(4):3–10, 1993.

[3] B.D. Carlstrom. *Embedding scheme in Java*. PhD thesis, 2001.

[4] Intel Corporation. Excerpts from a conversation with Gordon Moore: Moore's law. `ftp://download.intel.com/museum/Moores_Law/Video-Transcripts/Excepts_A_Conversation_with_Gordon_Moore.pdf`, 2005.

[5] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 238–252. ACM New York, NY, USA, 1977.

[6] P. Cousot and R. Cousot. Abstract interpretation frameworks. *Journal of logic and computation*, 2(4):511, 1992.

[7] P. Cousot and R. Cousot. Comparing the Galois connection and widening/narrowing approaches to abstract interpretation. In *Programming Language Implementation and Logic Programming*, pages 269–295. Springer, 1992.

[8] P.J. Denning and J.B. Dennis. The resurgence of parallelism. *Communications of the ACM*, 53(6):30–32, 2010.

[9] C. Flanagan and M. Felleisen. The semantics of future and its use in program optimization. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, page 220. ACM, 1995.

[10] C. Flanagan, A. Sabry, B.F. Duba, and M. Felleisen. The essence of compiling with continuations. *ACM SIGPLAN Notices*, 28(6):237–247, 1993.

[11] W.L. Harrison III. PARCEL: Project for the automatic restructuring and concurrent evaluation of lisp. In *Proceedings of the 2nd international conference on Supercomputing*, page 538. ACM, 1988.

[12] W.L. Harrison III. The interprocedural analysis and automatic parallelization of Scheme programs. *Lisp and Symbolic Computation*, 2(3):179–396, 1989.

[13] W.L. Harrison III and Z. Ammarguellat. Parcel and Miprac: parallelizers for symbolic and numeric programs. 1990.

[14] W.L. Harrison III and Z. Ammarguellat. A Program's Eye View of Miprac. In *Proceedings of the 5th International Workshop on Languages and Compilers for Parallel Computing*, pages 512–537. Springer-Verlag, 1992.

[15] P. Krajca and V. Vychodil. Data parallel dialect of scheme: outline of the formal model, implementation, performance. In *Proceedings of the 2009 ACM symposium on Applied Computing*, pages 1938–1939. ACM New York, NY, USA, 2009.

[16] E.A. Lee. The problem with threads. *Computer*, 39(5):33–42, 2006.

[17] Chuck C. Liang. Quicksort in scheme. `http://cs.hofstra.edu/~cscccl/csc123/quicksort.scm`, 2009.

[18] M. Might. *Environment analysis of higher-order languages*. PhD thesis, 2007.

[19] M. Might and T. Prabhu. Interprocedural Dependence Analysis of Higher-Order Programs via Stack Reachability. *Technical Report CPSLO-CSC-09-03*, page 75.

[20] M. Might and O. Shivers. Improving flow analyses via ΓCFA: Abstract garbage collection and counting. In *Proceedings of the 11th ACM International Conference on Functional Programming (ICFP 2006)*, pages 13–25, Portland, Oregon, September 2006.

[21] Matthew Might and Olin Shivers. Exploiting reachability and cardinality in higher-order flow analysis. *Journal of Functional Programming*, 18(5–6):821–864, 2008.

[22] J. Palsberg. Closure analysis in constraint form. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 17(1):47–62, 1995.

[23] PLT. Description of the gambit benchmarks, with comments. `http://racket-lang.org`, 2010.

[24] Todd Proebsting. Proebsting's law. `http://research.microsoft.com/en-us/um/people/toddpro/papers/law.htm`, 1998.

[25] The Larceny Project. Description of the gambit benchmarks, with comments. `http://www.larcenists.org/Twobit/benchmarksAbout.htm`, 2007.

[26] D. Schmidt. Abstract interpretation and static analysis. `http://santos.cis.ksu.edu/schmidt/Escuela03/WSSA/talk1p.pdf`, 2003.

[27] Michael I. Schwartzbach. Lecture notes on static analysis. 2008.

[28] O. Shivers. Control flow analysis in scheme. *ACM SIGPLAN Notices*, 23(7):174, 1988.

[29] O. Shivers. *Control-flow analysis of higher-order languages*. PhD thesis, 1991.

[30] G.L. Steele Jr. Rabbit: A compiler for Scheme. 1978.

[31] N.R. Tallent and J.M. Mellor-Crummey. Identifying Performance Bottlenecks in Work-Stealing Computations. *Computer*, 42(12):44–50, 2009.

[32] Wikipedia. Tak function. `http://en.wikipedia.org/wiki/Tak_(function)`, 2010.