1-1-2012

Using Dataflow Optimization Techniques with a Monadic Intermediate Language

Justin George Bailey
Portland State University

Recommended Citation

Bailey, Justin George, "Using Dataflow Optimization Techniques with a Monadic Intermediate Language" (2012). *Dissertations and Theses.* Paper 508.

10.15760/etd.508

Using Dataflow Optimization Techniques with a Monadic Intermediate Language

by

Justin George Bailey

A thesis submitted in partial fulfillment of the
requirements for the degree of

Master of Science
in
Computer Science

Thesis Committee:
Mark P. Jones, Chair
James Hook
Andrew Tolmach

Portland State University

Abstract

Our work applies the *dataflow algorithm* to an area outside its traditional scope: functional languages. Our approach relies on a *monadic intermediate language* that provides low-level, imperative features like computed jumps and explicit allocations, while at the same time supporting high-level, functional-language features like case discrimination and partial application. We prototyped our work in Haskell using the HOOPL library and this dissertation shows numerous examples demonstrating its use. We prove the efficacy of our approach by giving a novel description of the *uncurrying* optimization in terms of the dataflow algorithm, as well as a complete implementation of the optimization using HOOPL.

Acknowledgments

I wish to give my heartfelt thanks to Erin, my very patient and very understanding wife. She provided invaluable support throughout a long, long project. I also want to express my deep gratitude to my advisor, Dr. Mark P. Jones, who agreed to answer the question "So, how do functional language compilers work?" His ideas, enthusiasm, and willingness to mentor me made this project both challenging and highly rewarding.

I wish to thank my managers at ADP who gave me the time and freedom to pursue this and other projects: Mark Rankin, Tom Douglass, and Kathy Oullette. Without their support, I would simply have not been able to afford the time.

I want to give special thanks to my thesis committee, Dr. James Hook and Dr. Andrew Tolmach, for their time and willingness to review and comment on my work. Finally, my thanks to the other faculty and staff in the Computer Science department at Portland State University. I feel blessed that a place full of such smart, interesting and friendly people was just a little ways away from my home and work.

Contents

List of Figures

Chapter 1

Introduction

The *dataflow algorithm* treats programs as *control-flow* graphs, with edges representing execution paths and nodes representing statements or expressions. A particular dataflow analysis computes some desired property for each node in the graph based on a static approximation of the program's run-time behavior. The results of the analysis can then be used to optimize the program according to some measure, such as execution time, memory usage, or power consumption.

The first publication of the dataflow algorithm (Kildall, 1973) described a number of optimizations for programs written in ALGOL 60, an imperative programming language. Research and refinements since then have continued to focus on imperative languages.

In contrast, much research on the analysis and optimization of programs written in functional programming languages focuses on algebraic, rewrite-based techniques. This approach searches for syntactic patterns in the program's text and rewrites those patterns according to some set of rules.

No technical reason prevents the dataflow algorithm from being applied to functional language programs, but the technique has not been widely used. Tradition may play a role here, as well as pragmatic reasons. Rewriting programs according to syntactic patterns, when those programs are written as a set of "equational" definitions, seems much simpler than specifying transformations based on control-flow analysis.

Programs written in monadic style, as exemplified by the Haskell programming language, lend an imperative flavor to functional programs. Monadic programs typically impose explicit control-flow on the execution of the program, give the appearance of incrementally updating program variables, and allow imperative side-effects such as writing to the screen or reading input from the user.

This work explores the application of the dataflow algorithm to programs written in a *monadic intermediate language* (mil). Mil is a pure, functional language like Haskell, which requires all programs to be written in monadic style. We will show that dataflow analysis over mil programs can implement both functional-language specific and traditional imperative optimizations.

Chapter 2 gives a thorough introduction to the dataflow algorithm in its traditional setting. We explain the algorithm by applying *constant-propagation* to a simple C program. We introduce fundamental definitions used throughout this thesis, such as control-flow graphs and basic blocks. We discuss the theoretical basis of the algorithm, including its correctness and the quality of its solutions. Finally, we give the *dataflow equations* that can be used to describe any particular dataflow analysis.

In Chapter 3, we describe hoopl (Ramsey, Dias, and Peyton Jones, 2011), a Haskell implementation of the dataflow algorithm that we used to prototype all dataflow analyses described in this work. This chapter follows the structure of Chapter 2, emphasizing the connection between the theoretical description of the dataflow algorithm and hoopl's implementation. We use *dead-code elimination* for a subset of the C programming language as a running example.

We introduce our monadic intermediate language (MIL) in Chapter 4, describing its syntax, features, and design goals. We also introduce λ_C, a simple, high-level functional language that we use to define example programs that will be translated to MIL. We also discuss HOOPL's impact on the AST that we implemented to represent MIL programs.

Chapter 5 brings together the concepts introduced in previous chapters and describes the *uncurrying* optimization. We motivate the optimization, formulate it in terms of the dataflow algorithm, and show how it applies to MIL programs. We then describe how we implemented the optimization using HOOPL.

In Chapter 6 we discuss several ways in which this work could be extended. We describe how MIL programs can be optimized using only the *monad laws*. We briefly discuss dead-code elimination in MIL programs. We also sketch a more aggressive transformation of MIL programs that uses dataflow analysis of case alternatives to eliminate unnecessary allocations. We offer some reflections based on our experience with HOOPL, proposing ways in which we feel the library could be improved. Finally, we summarize our goals and the contributions of this work.

Chapter 2

Dataflow Optimization

In 1973, Gary Kildall described a framework for analyzing and transforming programs, calling it a *global analysis algorithm* (Kildall, 1973). His algorithm represents programs as *directed graphs*, where each node corresponds to a statement or expression in the program. The edges between nodes represent possible runtime execution paths. An *optimizing function* is applied to each node in the graph, transforming an *input pool* of facts into an *output pool*. When cycles occur in the graph, output pools can change input pools, causing the algorithm to apply the optimizing function again. His algorithm terminates when all output pools stop changing; the facts gathered can then be used to transform the program.

Though Kildall named his algorithm "global," he also applied it to smaller pieces of programs such as subroutines or function definitions. He showed that some analyses required reversing the input and output pools; in other words, running the algorithm backwards.

This chapter describes Kildall's algorithm, now known as *the dataflow algorithm* or the technique of *dataflow analysis*. In Section 2.1 we define *control-flow graphs* (CFGS), which the directed graphs representing the program are now called. Section 2.2 introduces "basic blocks," not something originally defined by Kildall but now a fundamental way of representing nodes in CFGS. We show the modern representation of the dataflow algorithm in Section 2.3, introducing terms and definitions that have been used since Kildall's original work. In Section 2.4 we show the general form of *dataflow equations* that can be used to describe any dataflow analysis; we will use these equations later in the thesis to describe our own analyses. Section 2.5 discusses the trustworthiness of the dataflow algorithm — that is, it shows how we can know a particular analysis has given the best possible solution. Transforming programs based on a dataflow analysis is discussed in Section 2.6, and we conclude in Section 2.7.

2.1 Control-Flow Graphs

Figure 2.1 shows a simple C program and its *control-flow graph* (CFG). Each *node* in Part (b) represents a statement or expression in the original program. For example, B_1 and B_2 represent the assignment statements on line 1. Notice that the declaration of `c` does not appear in a corresponding node; because the declaration does not cause a runtime effect, we do not represent it in the graph. Nodes E and X designate where program execution *enters* and *leaves* the graph. If the graph represented the entire program, we would say execution *begins* at E and *terminates* at X. However, the CFG may be embedded in a larger program, for which reason we say *enters* and *leaves*.

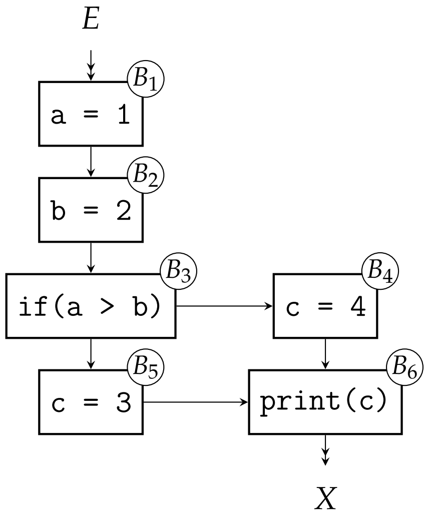
Directed edges show the order in which nodes execute. The edges leaving B_3 (representing the test "`if(a > b)`" on line 2) show that execution can branch to either B_4 (when $a > b$) or B_5 (when $a \leqslant b$). A node followed by multiple successors (i.e., where multiple edges leave the node) represents a *branch* or *conditional* statement. Any one of the successor nodes may execute following the conditional statement.

In this particular example, it is obvious that B_5 will always execute after B_3,

```
int a = 1, b = 2, c;
if(a > b)
  c = 4;
else
  c = 3;

print(c);
```

(a)



(b)

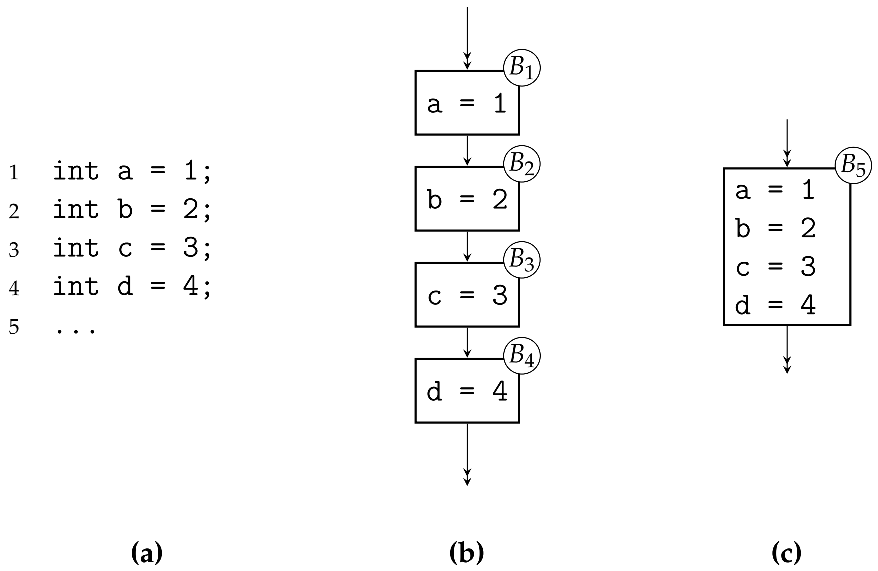
Figure 2.1: (a) A C-language program fragment. (b) The *control-flow graph* (CFG) for the program.

because the test will always fail. However, control-flow graphs show *possible* execution paths. They do not take into account the actual runtime values in a given graph. While in this case it is easy to determine how the program will behave, in general we cannot predict behavior without running the program.

The dataflow algorithm approximates a program's runtime state by analyzing the control flow of the program. Control-flow graphs show the order in which expressions and statements in a program are evaluated. It is the job of our *dataflow analysis* to determine how to make the program more efficient.

2.2 Basic Blocks

Consider the C-language fragment and control-flow graphs (CFG) in Figure 2.2. Part (b) shows the CFG for Part (a): a long, straight sequence of nodes, one after another. Part (c) represents the assignment statements on lines 1 – 4 as a *basic block*: a sequence of statements with one entry, one exit, and no branches in-between. Execution cannot start in the "middle" of the block, nor can it branch anywhere but at the end of the block. In fact, Part 2.2 (b) also shows four basic blocks — they just happen to consist of one statement each.



```
1  int a = 1;
2  int b = 2;
3  int c = 3;
4  int d = 4;
5  ...
```

Figure 2.2: (a): A C-language fragment to illustrate *basic blocks*. (b): The CFG for (a) without basic blocks. (c): The CFG for (c) using basic blocks.

The representation given in Part (c) has a number of advantages. It tends to reduce both the number of nodes and the number of edges in the graph. The dataflow algorithm maintains two sets of *facts* for every node — reducing the number of nodes obviously reduces the number of facts stored. The algorithm also iteratively propagates facts along edges — so reducing the number of edges reduces the amount of work we need to do. When rewriting, blocks allow us to move larger amounts of the program at once. It also can be shown (see Aho et al., 2006) that we do not lose any information by collapsing statements into blocks. For efficiency and brevity, we will work with basic blocks rather than statements from here forward.

2.3 The Dataflow Algorithm

Kildall's dataflow algorithm provides a general-purpose mechanism for analyzing control-flow graphs of programs. The algorithm itself does not mandate a specific analysis. Rather, it is parameterized by the choice of *facts*, *meet operator*, *transfer function*, and *direction*. The facts and meet operator form a lattice. Together, they approximate some property of the program that we wish to analyze. The transfer function transforms facts to mimic the flow of information in the control-flow graph. The direction is dictated by the type of analysis — each particular analysis runs *forwards* or *backwards*.

Consider Figure 2.3, Part (a), which shows a C function containing a loop that multiplies the argument by 10 some number of times. Line 2 declares `m` and assigns it the value 10. The function uses `m` in the loop body on Line 4 to repeatedly multiply the value passed in.

This function is just used for illustration — we do not expect anyone would actually write code this way (after all, `mult10` is just `10 * val * cnt`). In any case, the program in Figure 2.3 (a) can be transformed by replacing the variable `m` with 10 in the loop body. This may allow the compiler to generate code that directly multiplies `val` times 10 and saves using a register to hold the value of `m`. We can use a dataflow analysis known as *constant propagation* to justify this transformation.

```
1 int mult10(int cnt, int val) {
2   int m = 10, n = 0;
3   for(int i = 0; i < cnt; i++)
4     n += val * m;

5   return n;
6 }
```

(a)

```
1 int mult10(int cnt, int val) {
2   int m = 10, n = 0;
3   for(int i = 0; i < cnt; i++)
4     n += val * 10;

5   return n;
6 }
```

(b)

Figure 2.3: A C program which multiplies its argument, `val`, by 10 `cnt` times. Part (a) shows the original program. In Part (b), we have used *constant propagation* to replace the use of `m` in the loop body with 10.

The constant propagation analysis recognizes when a variable's value does not change in some context and then replaces references to the variable with the constant value. Figure 2.3, Part (b) shows the optimized program, replacing `m` with 10 on Line 4.

2.3.1 Facts and Lattices

Constant propagation determines if each variable's value changes during execution. The analysis *approximates* the actual values of the variables, as we cannot in general determine their exact value. We will place the value of each variable into one of three categories at each point in the control-flow graph: *unknown*, a *known integer constant*, or *indeterminate*. *Unknown*, represented by $\perp$ ("bottom"), is the initial value for all variables in our analysis. A *known integer constant*, $C \in \mathbb{Z}$, means our analysis identified that the variable was assigned a specific value that does not change. *Indeterminate*, indicated by $\top$ ("top"), means our analysis could not identify a constant value for the variable. Together, $\{\perp, \top\} \cup \mathbb{Z}$ forms a set which

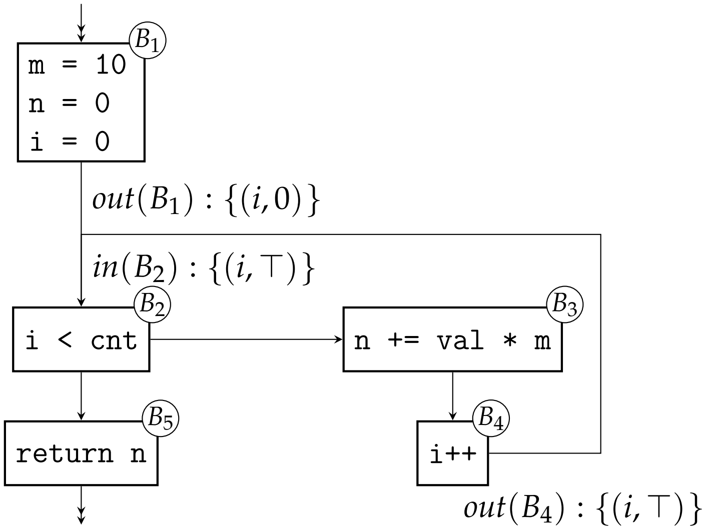


Figure 2.4: Our program, annotated with facts partway through the analysis. Notice that $out(B_1)$ and $out(B_4)$ give differing values to i. We use a *meet operator* when combining these two values while calculating $in(B_2)$.

we will denote as Const.

Figure 2.4 shows the control-flow graph of our program, annotated with *facts* about the variable i before and after nodes B_1, B_2 and B_4. This analysis defines facts as pairs, (a, x), where a is the name of a variable and $x \in$ Const. The *in* sets represent the value of the variables *before* the statement in the node executes; the *out* sets give the value of the variables afterwards. Together these sets represent our knowledge about each variable's value at each point in the program.

Constant propagation is a *forwards* analysis, so the values for each *in* set are calculated based on the *out* values of its predecessors. Figure 2.4 shows the facts computed partway through this analysis, concentrating on the nodes that reference i: B_1, B_2 and B_4. B_2 has two predecessors: B_1 and B_4. Their *out* sets, $out(B_1)$

and *out*(B_4), give differing values to i: 0 and $\top$. To combine these values when computing *in*(B_2), we use a *meet operator*.

The *meet operator*, $\sqcap$, defines how we will combine values in Const. Figure 2.5 gives the definition of $\sqcap$. For any value of $x \in$ Const, $\bot \sqcap x$ results in x. Conversely, $x \sqcap \top$ results in $\top$. Two differing constants, C_1 and C_2, result in $\top$, while equal constants give the same constant.

v_1	v_2	$v_1 \sqcap v_2$	
$\bot$	x	x	
x	$\top$	$\top$	
C_1	C_2	$\top$	$(C_1 \neq C_2)$
C_1	C_2	C_1	$(C_1 = C_2)$

Figure 2.5: Definition of the *meet operator*, $\sqcap$, for the lattice used in our constant propagation analysis. v_1 and v_2 are values in Const. The table shows how $\sqcap$ combines any two values.

Our definition of $\sqcap$ allows us to distinguish variables for which we have no information from those with non-constant values. This matches the definition of the C language, where a variable that is not initialized has an undefined value. This allows us to legally transform some programs by assuming the undefined variable has the best possible value. For example, Figure 2.6 shows a block with two predecessors. *Out*(B_1) has the fact $(x, 1)$, while *out*(B_2) says $(x, \bot)$. Our definition of $\sqcap$ intuitively says that, when we do not know anything about a variable, we can use the information we already have instead. *Out*(B_2) adds no information about x; we can assume it is 1, the best possible value. Our definition of $\sqcap$ follows that intuition and tells us that *in*(B_3) should be $(x, 1 \sqcap \bot)$, or $(x, 1)$.

We would not make this assumption if we wished to warn the programmer that

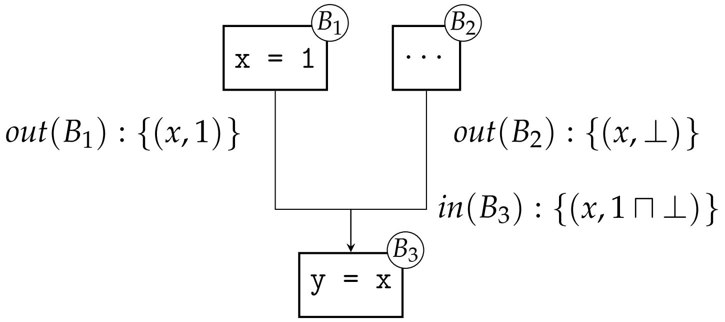

x = 1
B_1
...
B_2
$out(B_1) : \{(x, 1)\}$
$out(B_2) : \{(x, \perp)\}$
$in(B_3) : \{(x, 1 \sqcap \perp)\}$
y = x
B_3

Figure 2.6: A control-flow graph illustrating the behavior of $\sqcap$ with $\perp$ (i.e., undefined) values.

a potentially uninitialized variable could be used. In that case, we would define $\sqcap$ such that $x \sqcap \perp$ was $\perp$. Then, when the fact $(x, \perp)$ appeared, we could warn that a variable might be used before being initialized. Similarly, if our language defined an initial value for all variables, our assumption would have no effect. We could use the same definition, but no variable would have the value $\perp$ — each would have a known initial value.

We have defined $\sqcap$ on elements in Const, but our facts are pairs (a, x) where a is a variable and x a value in Const; *in* and *out* are sets of facts. Therefore, we define the $\wedge$ ("wedge") operator to apply $\sqcap$ to sets of facts (F_1 and F_2 below):

$$\begin{aligned} F_1 \wedge F_2 = &\{(a, x_1 \sqcap x_2) \mid (a, x_1) \in F_1, (a, x_2) \in F_2\} \\ &\cup \{(a, x_1) \mid (a, x_1) \in F_1, a \notin dom(F_2)\} \\ &\cup \{(a, x_2) \mid (a, x_2) \in F_2, a \notin dom(F_1)\} \qquad (2.1) \\ dom(F) = &\{a \mid (a, x) \in F\} \qquad (2.2) \end{aligned}$$

Our $\wedge$ operator acts like union when a variable in F_1 does not appear in the domain of F_2; likewise for a variable only in F_2. When a variable appears in both F_1 and F_2, the values for the variable are combined using $\sqcap$.

To compute $in(B)$, we apply $\wedge$ to each *out* set of B's predecessors. We use a subscripted $\bigwedge$ to indicate we combine all of the *out* sets into one using $\wedge$:

$$in(B) = \bigwedge_{P \in pred(B)} out(P). \tag{2.3}$$

With these definitions, we can now show how the $in(B_2)$ set in Figure 2.4 is derived:

$$\begin{aligned}
in(B_2) &= \bigwedge_{P \in pred(B_2)} out(P) \\
&\qquad\qquad \textit{Predecessors of } B_2\textit{; Equation } (2.3). \\
&= out(B_1) \wedge out(B_4) \\
&\qquad\qquad \textit{Definition of } out(B_1) \textit{ and } out(B_4). \\
&= \{(i, 0)\} \wedge \{(i, \top)\} \\
&\qquad\qquad \textit{Equation } (2.1). \\
&= \{(i, 0 \sqcap \top)\} \\
&\qquad\qquad \textit{Definition of } \sqcap \textit{ from Figure 2.5.} \\
&= \{(i, \top)\} \\
&\qquad\qquad \textit{Definition of } in(B_2). \\
&= \{(i, \top)\}.
\end{aligned}$$

Together, Const and $\sqcap$ form a *lattice*.[1] The lattice precisely defines the facts computed in our analysis. In this case, the lattice represents knowledge about a variable's value. Each specific dataflow analysis computes different facts, but those facts are always represented by a lattice.

2.3.2 Transfer Functions

The dataflow algorithm calculates new facts using a *transfer function*. The transfer function is specific to both the analysis performed and the semantics of the source language for the programs that are analyzed. In principle, each node in the graph

[1] The lattice can also have a *join* operator, but for our purposes we solely use the meet.

can have its own transfer function, but in practice the function is defined by cases for each statement or expression in the language.

In a *forwards* analysis, the transfer function computes the *out* set for a given node. In a backwards analysis, the transfer function computes the *in* set. That is, a forwards analysis computes facts that hold *after* a node executes. A backwards analysis computes facts that were true *before* a node executed. In both cases, the transfer function also considers known facts (i.e., *in* facts for forwards, *out* for backwards) as well as the statements in the node.

For our example analysis, only two kinds of statements can affect the facts we calculate: constant and non-constant updates. A constant update is one of the form $a = C$, where C is a known integer value. A non-constant update is any other type of assignment; in our example, something like *i*++. Any other type of statement will have no effect on our facts.

We define a transfer function, t, for our analysis in terms of statements in our source language. Our function takes a set of input facts (F), and a statement; it produces a set of output facts:

$$\begin{aligned} t(F, a \texttt{ = } C) &= \{(a, C)\} \cup (F \setminus \mathit{uses}(F, a)) \\ t(F, a\texttt{++}) &= \{(a, \top)\} \cup (F \setminus \mathit{uses}(F, a)) \\ t(F, a \texttt{ += } b) &= \{(a, \top)\} \cup (F \setminus \mathit{uses}(F, a)) \\ t(F, s) &= F \end{aligned} \tag{2.4}$$

$$\mathit{uses}(F, a) = \{(a', x) \mid (a', x) \in F \text{ and } a = a'\}.$$

When a node contains a constant update ($a = C$), then Equation (2.4) adds that fact to the input set. For a non–constant update, a new fact $(a, \top)$ is always added to the output set. In both cases, all mentions of a in the input set are removed before being combined with the new fact — this ensures that no more than one fact per variable exists in our fact set.

The definition in Equation (2.4) matches our intuition for constant propagation. When we know a variable is assigned a constant, we add that fact to our knowledge. When we know it is changed in a non-constant way, we update our knowledge to show we no longer know the value of the variable. If the statement is not an assignment, we leave the facts unchanged.

Figure 2.7, Part (a), shows our program, annotated with initial *in* and *out* facts. Figure 2.7, Part (b), shows the same graph with annotations updated using Equation (2.4). The assignments in B_1 create the facts $(m, 10)$, $(n, 0)$, and $(i, 0)$ in $out(B_1)$. The assignment to n in B_3 is a non-constant update so $out(B_3)$ contains $(n, \top)$. Similarly, the increment to i in B_4 creates the fact $(i, \top)$ in $out(B_4)$.

Notice that the updated *out* sets do not affect subsequent *in* sets. $In(B_3)$ does not contain the fact $(m, 10)$ from $out(B_1)$. $In(B_2)$ also does not show the facts $(i, \top)$ or $(i, 0)$ from either $out(B_4)$ or $out(B_2)$. The next section on iteration will discuss how we update *in* sets as *out* sets change.

2.3.3 Iteration & Fixed Points

Figure 2.7 hints that facts develop over time during analysis. In fact, the transfer function is applied to each node in turn, creating new facts from old, until the facts stop changing. In other words, the control-flow graph is analyzed *iteratively* until all *out* (in the case of a forwards analysis) or *in* (in the case of a backwards analysis) sets reach a *fixed point*.

Figure 2.8, Part (a) shows how the *in* and *out* sets for each node change during our analysis. The "zeroth" iteration corresponds to the initial value for all facts: everything is $\bot$ (i.e., unknown). Reading from left-to-right gives the *in* and *out*

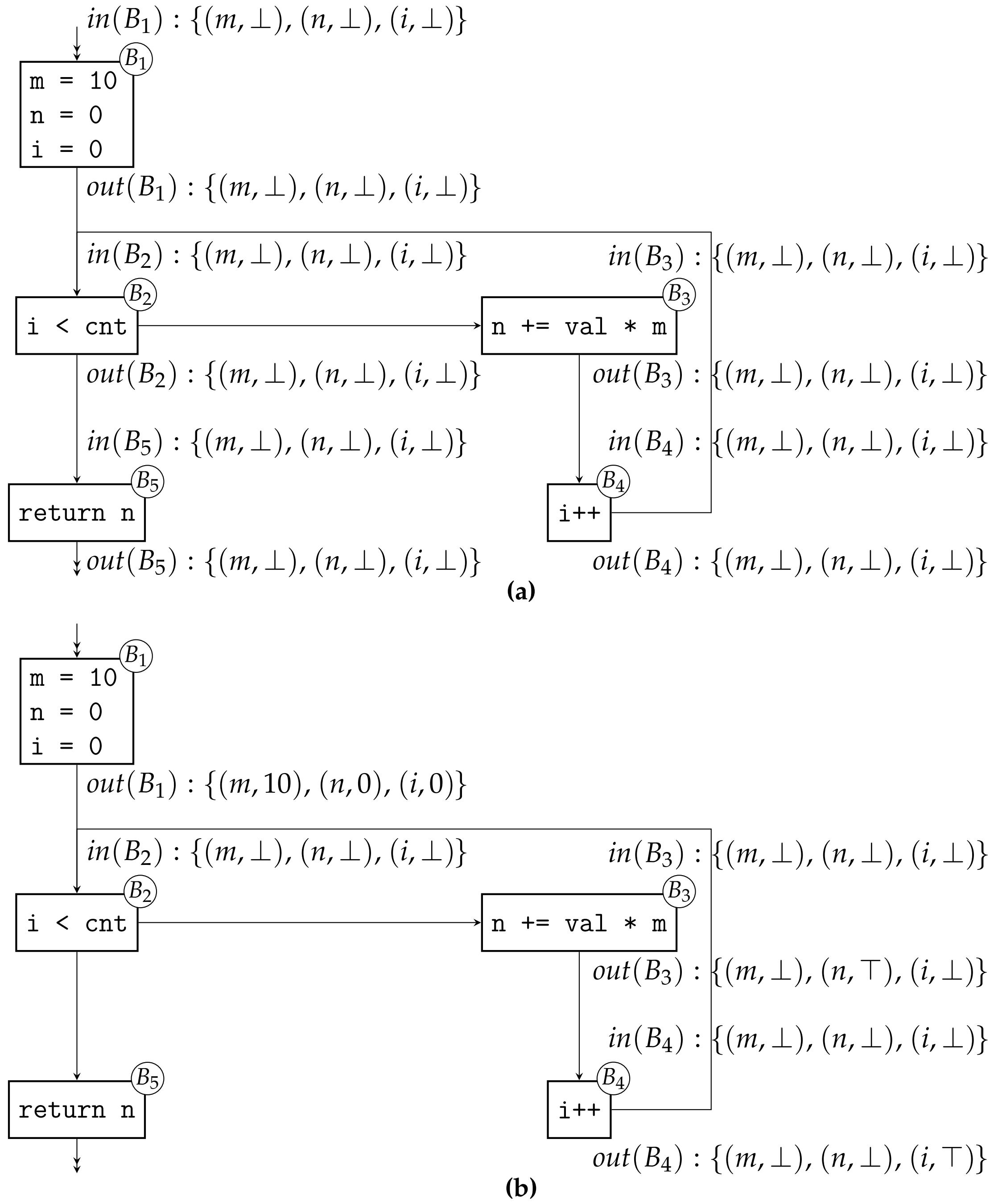
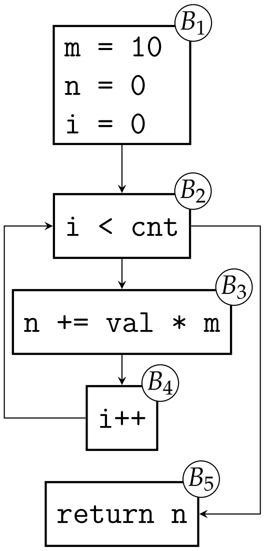

$in(B_1) : \{(m, \bot), (n, \bot), (i, \bot)\}$
B_1
m = 10
n = 0
i = 0
$out(B_1) : \{(m, \bot), (n, \bot), (i, \bot)\}$
$in(B_2) : \{(m, \bot), (n, \bot), (i, \bot)\}$
$in(B_3) : \{(m, \bot), (n, \bot), (i, \bot)\}$
B_2
i < cnt
B_3
n += val * m
$out(B_2) : \{(m, \bot), (n, \bot), (i, \bot)\}$
$out(B_3) : \{(m, \bot), (n, \bot), (i, \bot)\}$
$in(B_5) : \{(m, \bot), (n, \bot), (i, \bot)\}$
$in(B_4) : \{(m, \bot), (n, \bot), (i, \bot)\}$
B_5
return n
B_4
i++
$out(B_5) : \{(m, \bot), (n, \bot), (i, \bot)\}$
$out(B_4) : \{(m, \bot), (n, \bot), (i, \bot)\}$
(a)
B_1
m = 10
n = 0
i = 0
$out(B_1) : \{(m, 10), (n, 0), (i, 0)\}$
$in(B_2) : \{(m, \bot), (n, \bot), (i, \bot)\}$
$in(B_3) : \{(m, \bot), (n, \bot), (i, \bot)\}$
B_2
i < cnt
B_3
n += val * m
$out(B_3) : \{(m, \bot), (n, \top), (i, \bot)\}$
$in(B_4) : \{(m, \bot), (n, \bot), (i, \bot)\}$
B_5
return n
B_4
i++
$out(B_4) : \{(m, \bot), (n, \bot), (i, \top)\}$
(b)

Figure 2.7: Part (a) shows our program annotated with initial facts. In Part (b), we have updated each $out(B)$ set using Equation (2.4), our transfer function.

Iteration:	0	1	2	3	4
$in(B_1)$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$
$out(B_1)$	$\bot$	0	0	0	0
$in(B_2)$	$\bot$	$\bot$	$\top$	$\top$	$\top$
$out(B_2)$	$\bot$	$\bot$	$\top$	$\top$	$\top$
$in(B_3)$	$\bot$	$\bot$	$\bot$	$\top$	$\top$
$out(B_3)$	$\bot$	$\bot$	$\bot$	$\top$	$\top$
$in(B_4)$	$\bot$	$\bot$	$\bot$	$\bot$	$\top$
$out(B_4)$	$\bot$	$\top$	$\top$	$\top$	$\top$
$in(B_5)$	$\bot$	$\bot$	$\bot$	$\top$	$\top$
$out(B_5)$	$\bot$	$\bot$	$\bot$	$\top$	$\top$

(a)



(b)

Figure 2.8: This figure shows the values for *i* calculated by all nodes in our example program. Part (a) shows the *in* and *out* facts associated with each node, for variable *i*. Part (b) reproduces the control-flow graph for our program. After 4 iterations the facts reach a fixed point (i.e., they stop changing).

facts for a given node at each iteration of the analysis. The control-flow graph is reproduced in Part (b). Following the control-flow between nodes shows which *out* sets are used to calculate *in* sets.

Consider the value for *i* in $in(B_2)$, the node that tests the condition `i < cnt`. In the first iteration, $in(B_2)$ still assigns $\bot$ to *i*. Equation (2.3) states that $in(B_2)$ is derived from the *out* sets of B_2's predecessors: B_1 and B_4. By Equation (2.3) we can calculate the value of *i* in $in(B_2)$. Crucially, the *out* set used comes from the *previous* iteration of the analysis, which we emphasize by attaching the iteration number to each set:

$$\begin{aligned} in(B_2)^1 &= \bigwedge_{P \in pred(B)} out(P). \\ &= out(B_1)^0 \bigwedge out(B_4)^0 \\ &= \{\ldots, (i, \bot), \ldots\} \wedge \{\ldots, (i, \bot), \ldots\} \\ &= \{\ldots, (i, \bot \sqcap \bot), \ldots\} \\ &= \{\ldots, (i, \bot), \ldots\}. \end{aligned}$$

Now consider the second iteration, where $in(B_2)$ assigns $\top$ to i. $out(B_1)$ gives i the value 0 (by `i = 0`). However, $out(B_4)$ assigns i the value $\top$, because `i++` is a non-constant update. We can see why $(i, \top) \in in(B_2)$ by Equation (2.3). Again we attach the iteration number to each set, emphasizing its origin:

$$\begin{aligned} in(B_2)^2 &= out(B_1)^1 \bigwedge out(B_4)^1 \\ &= \{\ldots, (i, 0), \ldots\} \wedge \{\ldots, (i, \top), \ldots\} \\ &= \{\ldots, (i, 0 \sqcap \top), \ldots\} \\ &= \{\ldots, (i, \top), \ldots\}. \end{aligned}$$

Notice how the conflicting values for i are resolved with the $\sqcap$ operator. The value of i in $out(B_2)$ has reached a fixed point with this iteration; it will no longer change.

The above example raises an important question: how do we know that our analysis will terminate? Will the algorithm iterate endlessly over the cfg, updating facts and never stopping? The answer is that the dataflow algorithm will terminate if our lattice has *finite height* and a *monotone* transfer function.

Let us begin with the lattice. Consider again the meet operator, $\sqcap$, defined in Figure 2.5 and our set of values, Const:

$$\textsc{Const} = \{\bot, \top\} \cup \mathbb{Z}.$$

$$\begin{aligned}
\bot \sqcap x &= x. \\
x \sqcap \top &= \top. \\
C_1 \sqcap C_2 &= \top, \text{where } C_1 \neq C_2. \\
C_1 \sqcap C_1 &= C_1.
\end{aligned}$$

The definition of $\sqcap$ imposes a *partial order* on the values in Const. That is, we can define an operator, $\sqsubseteq$, such that for all x and y in Const:

$$x \sqsubseteq y \text{ if and only if } x \sqcap y = y. \tag{2.5}$$

That is, x is "less than or equal to" y only when $x \sqcap y$ equals y.

We now define the *height* of our lattice as the longest possible ordering of values in Const such that:

$$x_1 \sqsubseteq x_2 \ldots \sqsubseteq x_n, \text{where } x_1 \neq x_2 \neq \cdots \neq x_n. \tag{2.6}$$

That is, the height is the longest possible path from the "lowest" to "highest" element of the lattice where we do not repeat any values and where the $\sqsubseteq$ relation holds among all values.

We can more succinctly define the height using a strict "less than" ordering. First, the $\sqsubset$ relation:

$$x \sqsubset y \text{ if and only if } x \sqcap y = y \text{ and } x \neq y. \tag{2.7}$$

And now we can redefine height as the largest n such that:

$$x_1 \sqsubset x_2 \ldots \sqsubset x_n. \tag{2.8}$$

We can show by contradiction that the height of our lattice is 3. Suppose there exists $x_1 \sqsubset x_2 \sqsubset x_3 \sqsubset x_4$. If x_4 is $\top$, then x_3 must be an integer or $\bot$. If x_3 is $\bot$, by Equation (2.8), there is no such x_2 such that $x_2 \sqsubset \bot$. Therefore, x_3 cannot be $\bot$. If x_3 is an integer, again by Equation (2.8), x_2 must be $\bot$. In turn, there is no such x_1 such that $x_1 \sqsubset \bot$. Therefore, x_4 cannot be $\top$ and in fact, by similar arguments, it cannot exist. But we know that, for all $C \in \mathbb{Z}$, $\bot \sqsubset C \sqsubset \top$, which is a path of length 3, so it follows that the height of our lattice must be 3.

Now let us address the transfer function. A *monotone* function has the following property:

$$f(x) \sqsubseteq f(y) \text{ whenever } x \sqsubseteq y. \tag{2.9}$$

That is, if x is "less than or equal to" y, $f(x)$ will also be "less than or equal to" $f(y)$.

The transfer function moves our facts along the lattice. The lattice represents the information we have gathered during our analysis. In turn, the ordering of values represents "how much" we know. That is, when a variable is assigned $\bot$, we do not know anything about it. If it is assigned $\top$, we have seen "too many" assignments (or some other update). A monotone transfer function always increases (or does not change) the information we have. For all statements, our transfer function either does not change the set of facts or it updates them so a given variable is either associated with some $C \in \mathbb{Z}$ or the $\top$ value. Therefore, our transfer function must be monotone.

2.4 Dataflow Equations

As stated at the beginning of this chapter, the dataflow algorithm specifies four parameters: facts, meet operator, transfer function, and direction. The prior section presented each parameter for the constant propagation analysis separately. Figure 2.9 presents all of them together, as a set of *dataflow equations*. Pairs of elements from Const and Var define Fact, our set of facts. Equation (2.11) defines our meet operator, $\wedge$, on Fact values. Our transfer function t, defined by Equation (2.12), shows how we create new facts based on the statements in each node and our existing facts. Equation (2.14) shows that we compute $out(B)$ using the transfer function t and the $in(B)$ facts for the block. Equation (2.15) states that we apply $\wedge$ to all of the *out* sets for the predecessors of a block B in order to calculate $in(B)$. Together, Equations (2.14) and (2.15) specify a forwards dataflow analysis.

We can define an iterative dataflow algorithm in terms of these parameters. Figure 2.10 gives the algorithm for a forwards analysis.[2] On Line 1, we initialize all *out* and *in* sets to some suitable initial value from Fact. The superscript on *in* and *out* sets refer to sets from the i^{th} iteration; initialization constitutes the "zeroth" iteration. Sometimes the entry node's *out* set gets special treatment, in which case we could add the line:

```
Out(Entry)^0 = v, v ∈ Fact.
```

However, $out(Entry)$ normally gets the same value as other *out* sets.

[2] The backwards case is almost identical.

Facts

$$\begin{aligned}
\textsc{Const} &= \{\bot, \top\} \cup \mathbb{Z}. \\
\textsc{Var} &= \text{Set of all variables.} \\
\textsc{Fact} &= \textsc{Var} \times \textsc{Const}.
\end{aligned}$$

Meet

$$\begin{aligned}
F_1 \wedge\ F_2 \;=\; & \{(a, x_1 \sqcap x_2) \mid (a, x_1) \in F_1, (a, x_2) \in F_2\} \\
& \cup \{(a, x_1) \mid (a, x_1) \in F_1, a \notin dom(F_2)\} \\
& \cup \{(a, x_2) \mid (a, x_2) \in F_2, a \notin dom(F_1)\}, \\
& \text{where } F_1, F_2 \in \textsc{Fact}, \sqcap \text{ as in Figure 2.5, } dom \text{ from Equation (2.2).}
\end{aligned} \tag{2.11}$$

Transfer

$$\begin{aligned}
t(F, a \texttt{ = } C) &= \{(a, x \sqcap C), \text{when } (a, x) \in F \text{ or} \\
&\qquad (a, C), \text{when } a \notin dom(F)\} \cup \\
&\quad F \setminus uses(F, a), \\
&\text{where } F \in \textsc{Fact}, C \in \mathbb{Z}. \\
t(F, a\texttt{++}) &= \{(a, \top)\} \cup (F \setminus uses(F, a)), \\
&\text{where } F \in \textsc{Fact}. \\
t(F, a \texttt{ += } b) &= \{(a, \top)\} \cup (F \setminus uses(F, a)). \\
&\text{where } F \in \textsc{Fact}.
\end{aligned} \tag{2.12}$$

$$\begin{aligned}
uses(F, a) &= \{(a, x) \mid a \in dom(F)\}, \\
&\text{where } F \in \textsc{Fact}, a \in \textsc{Var}.
\end{aligned} \tag{2.13}$$

Direction

$$out(B) = t(in(B), s), \tag{2.14}$$

where s a statement in block B.

$$in(B) = \bigwedge_{P \in pred(B)} out(P) \tag{2.15}$$

$$pred(B) = \text{Predecessors of block } B.$$

Figure 2.9: The transfer function and associated definitions for the constant propagation analysis. Equation (2.14) shows how *out* facts are created from *in* facts. *In*(B) facts, for some block B, are created from the *out*(B) facts of its predecessors. Facts are combined using the set-wise $\bigwedge$ operator.

```
1 in$(B)^0$ = $u$, out$(B)^0$ = $f$, $\forall$ nodes $B$; $f$, $u$ $\in$ Fact
2 do {
3   in$(B)^{i+1}$ = $\bigwedge_{P \in pred(B)}$ out$^i(P)$
4   out$(B)^{i+1}$ = $t$(in$(B)^i$, $B$)
5 } until out$(B)^{i+1}$ = out$(B)^i$, $\forall$ $B$
```

Figure 2.10: The dataflow algorithm, using parameters for facts, the meet operator, direction, and the transfer function.

The main loop of the algorithm always executes at least once. On Line 3, we calculate *in* facts for each node B in the next iteration, $in(B)^{i+1}$, by applying $\wedge$ to the out^i sets of B's predecessors from the current iteration. Line 4 calculates $out(B)^{i+1}$ for each node by applying the transfer function, t, to that node, along with $in(B)^i$, the *in* facts for the current iteration.

Line 5 checks if all $out\ ^{i+1}$ sets are equal to their previous value, out^i. If not, the loop repeats. Otherwise the algorithm terminates. The final values for each $out(B)$ set then hold the facts representing the result of our analysis.

We have presented the iterative, forwards dataflow algorithm and shown how the algorithm can be parameterized for a particular analysis. We gave the parameterization for our constant propagation analysis in Figure 2.9. We know the algorithm will terminate if our transfer function is *monotone* and our lattice has *finite* height. However, we have not discussed how to measure the results our analysis gives us — how do we know that they are the best possible? We will address that question in the next section.

2.5 Quality of Solutions to the Dataflow Equations

Aho (Aho et al., 2006) shows that for a dataflow analysis defined with a finite lattice and monotone transfer function, Figure 2.10 will compute a *maximum fixed point*. A maximum fixed point means that for any $F = out(B)$ computed by the algorithm, no other F' can be computed such that $F \sqsubset F'$. In other words, the process described in Figure 2.10 will compute *out* facts with the best possible information that our algorithm is capable of.

The maximum fixed point solution differs from the *ideal* solution in that the maximum fixed point solution may make more conservative estimates than necessary. In particular, the algorithm does not consider knowledge about branches that will never be taken. For example, the C program from Figure 2.1 (a) will never execute Line 3, because the test `if(a > b)` is always false:

```
1  int a = 1, b = 2, c;
2  if(a > b)
3    c = 4;
4  else
5    c = 3;
6  ...
```

Our algorithm, however, does not take such conditions into account. The ideal solution considers only the paths that will be taken by the program. Determining the actual paths taken is an undecidable problem — thus we settle for the maximum fixed point. Fortunately, the algorithm is conservative — it never ignores (or adds) paths — so we can be sure that its analysis will never be wrong, just that it may not be as good as the ideal.

2.6 Applying Results

Figure 2.3, Part (a) on Page 9 gave a sample program that we wished to optimize using a *constant propagation* dataflow analysis. Figure 2.3, Part (b) gave the result, replacing all occurrences of m with 10. Now, knowing the dataflow algorithm and the equations for constant propagation, we can derive how that transformation is made.

Figure 2.11 gives the facts calculated for all nodes in our program, during each iteration of the analysis. The first iteration calculates that $out(B_1)$ assigns m the value 10, due to the assignment `m = 10` on Line 2. The second iteration propagates this value to $in(B_2)$ and in turn to $out(B_2)$, because the test on Line 3 does not affect m. In the third iteration, we see the same with $in(B_3)$ and $out(B_3)$ on Line 4. The analysis continues for two more iterations as other values propagate, but at this point we have all the information we need to optimize the program. Once the analysis reaches a fixed point, we can safely replace all occurrences of m with 10, resulting in the optimized program given in Figure 2.3, Part (b).

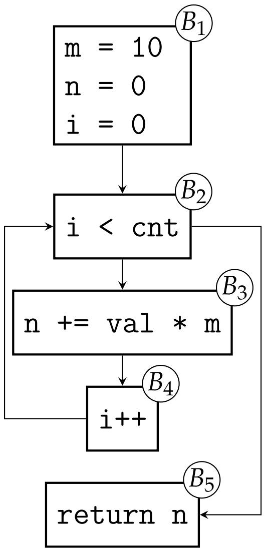
We have now seen how we can use constant propagation to optimize a simple program. Typically many more optimizations will be run over the same code, each (hopefully) improving it a little more. For example, we could use an optimization called *dead-code elimination* to remove the declaration of m altogether from our optimized program, as it is no longer used.

2.7 Summary

This chapter gave an overview of *dataflow optimization*. The dataflow *algorithm* gives a general technique for applying an *optimizing function* to the *control flow graph* (cfg)

Iteration:	0			1			2			3			4			5		
	m	*n*	*i*	*m*	*n*	*i*	*m*	*n*	*i*	*m*	*n*	*i*	*m*	*n*	*i*	*m*	*n*	*i*
$in(B_1)$	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥
$out(B_1)$	⊥	⊥	⊥	10	0	0	10	0	0	10	0	0	10	0	0	10	0	0
$in(B_2)$	⊥	⊥	⊥	⊥	⊥	⊥	10	0	⊤	10	⊤	⊤	10	⊤	⊤	10	⊤	⊤
$out(B_2)$	⊥	⊥	⊥	⊥	⊥	⊥	10	0	⊤	10	⊤	⊤	10	⊤	⊤	10	⊤	⊤
$in(B_3)$	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	10	0	⊤	10	⊤	⊤	10	⊤	⊤
$out(B_3)$	⊥	⊥	⊥	⊥	⊤	⊥	⊥	⊤	⊥	10	⊤	⊤	10	⊤	⊤	10	⊤	⊤
$in(B_4)$	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊤	⊥	⊥	⊤	⊥	10	⊤	⊤	10	⊤	⊤
$out(B_4)$	⊥	⊥	⊥	⊥	⊥	⊤	⊥	⊤	⊤	⊥	⊤	⊤	10	⊤	⊤	10	⊤	⊤
$in(B_5)$	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	10	0	⊤	10	⊤	⊤	10	⊤	⊤
$out(B_5)$	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	10	0	⊤	10	⊤	⊤	10	⊤	⊤

(a)



(b)

Figure 2.11: This figure shows the facts calculated for all nodes in our example program. Part (a) shows the *in* and *out* facts associated with each node. Part (b) reproduces the control-flow graph for our program. After 5 iterations the facts reach a fixed point (i.e., they stop changing) and we can see that $in(B_3)$ shows that m is always 10, proving we can rewrite the multiplication safely.

representing a given program. The optimizing function computes *facts* about each node in the graph, using a *transfer* function. A given analysis can proceed *forwards* (where $in(B)$ facts are used to produce $out(B)$ facts) or *backwards* (where $out(B)$ facts are used to produce $in(B)$ facts). Each optimization defines a specific *meet operator* that combines facts for nodes with multiple predecessors (for forwards analysis) or successors (for backwards). We compute facts *iteratively*, stopping when they reach a *fixed point*. Finally, we *rewrite* the CFG using the facts computed as a guide. The meaning of our program does not change, but its behavior may be "better," whatever that means for the particular optimization applied.

Chapter 3

The Hoopl Library

The dataflow algorithm describes a method for analyzing programs based on the computation of facts between nodes in the program's control-flow graph. The HOOPL library (Ramsey, Dias, and Peyton Jones, 2011), written in Haskell, provides a framework for using the dataflow algorithm. Hoopl enables the user to implement their own analyses for their own programming language. A thorough description of the library's implementation can be found in the authors' paper (Ramsey, Dias, and Peyton Jones, 2010); here, we discuss the abstractions they provide and how to use them.

Hoopl's implementation follows a variation of the dataflow algorithm described by Lerner, Grove and Chambers (2002). In brief, Lerner and colleagues' dataflow algorithm interleaves analysis and transformation. While this technique does not provide any better quality solutions than Kildall's original formulation (1973), it makes the composition of multiple dataflow analyses much simpler. We return to Lerner and colleagues' work in Section 3.6.

Hoopl implements the generic portions of the dataflow algorithm: iterative computation, traversing the control-flow graph (CFG), and combining facts. The *client program*, a term HOOPL uses to mean the program using the library for some optimization, provides data structures and functions specific to that optimization: the representation of programs and facts, a transfer function, a rewriter, and a

```
1 void example() {
2   int a, c;
3   c = 4;
4   a = c + 1;
5   printf("%d",c);
6   a = c + 2;
7 }
```

(a)

```
1 void example() {
2   int c;
3   c = 4;
4   printf("%d",c);
5 }
```

(b)

Figure 3.1: Part (a) defines a function using the C language. Part (b) shows the program after performing dead-code elimination.

meet operator.

We will illustrate hoopl concepts using a running example motivated by the C-language function defined in Figure 3.1 (a). A cursory examination of that listing shows the assignments to `a` on Lines 4 and 6 do not affect the output (i.e., observable behavior) of `example`. We could eliminate them without changing the program's meaning; we may even improve its performance. However, we could not eliminate the assignment to `c` on Line 3 because that may change the value printed on Line 5. We call variables that may affect observable behavior *live*; a *dead* variable is not live. Figure 3.1 (b) shows one way we could optimize this program by eliminating the "dead" statements in Figure 3.1 (a).

Dead-code elimination refers to the optimization that first determines "liveness" and then removes dead statements (i.e., those only assigning to dead variables). Our running example will implement a client program that can apply dead-code elimination to the program in Figure 3.1 (a), transforming it to resemble Figure 3.1 (b).

This chapter provides enough background to understand the use of hoopl in this work. It assumes the reader has prior knowledge of the Haskell programming language, including language extensions such as GADTs (Schrijvers et al., 2009), as implemented by GHC 7.2 (The GHC Team, 2011b). This chapter follows the same outline as our chapter covering dataflow analysis, presenting each concept in terms of HOOPL structures. Section 3.1 gives an overview of the types, data structures, and functions provided by HOOPL. Sections 3.2 through 3.8 give detailed information about each item. Throughout, we develop our client program to implement dead-code elimination. We conclude with a summary of HOOPL in Section 3.9.

3.1 Hoopl's API

To implement dataflow analysis generically, HOOPL defines several core data structures that client programs must use. These include the representation of CFGs, the types of transfer and rewrite functions, and the implementation of the meet operator. Hoopl controls the CFG representation so it can traverse, propagate facts around, and rewrite the CFG. Hoopl specifies the type of the transfer and rewrite function such that they produce usable information (and rewrites). Finally, HOOPL specifies the form of the meet operator (but not its implementation) so that the library can recognize fixed points.

Hoopl requires that client programs specify those items related to their specific optimization: the abstract syntax tree (AST) of the language analyzed, the representation of facts, and the implementation of the transfer and rewrite functions. Each node in the CFG typically contains an expression or statement from the AST of the language that the client program analyzes. While HOOPL controls the edges between nodes in the CFG, it does not specify the contents of those nodes. Similarly, while HOOPL determines when an analysis reaches a fixed point, HOOPL relies on the client to specify when one set of facts equals another. Finally, HOOPL applies the transfer and rewrite functions to the CFG but requires that the client program implement them for their specific AST and optimization.

3.2 Control-Flow Graphs

Hoopl defines CFGs in terms of basic blocks, parameterized by *content* and *shape*. Content takes the form of statements or expressions from the client's AST, while shape specifies how control-flow enters and leaves a given block. An "open" block allows control-flow to fall-through implicitly from its predecessor or to fall-through to its successor. A "closed" block requires that control-flow explicitly transfers to or from the block. Shape constrains the CFG such that only blocks with compatible shapes can be connected: predecessors of an open block must be open; successors of a closed block must be closed.

Hoopl provides types named O (for open) and C (for closed) to describe the entry and exit shape of a given block. We write O O ("open/open"), O C ("open/closed"), etc., where the first type describes the block's entry shape and the latter its exit shape. An O C block requires a unique predecessor. Control-flow will fall-through from the predecessor to the O C block, but control-flow must explicitly transfer to a successor block on exit. An O O block requires a unique predecessor and a unique successor; this allows control-flow to fall-through from its predecessor and similarly allows control-flow to implicitly pass to its successor. A C O block requires that control-flow passes explicitly from its predecessors. However, control-flow falls-through from the block to its successor. A C C block must be the target of an explicit control-flow transfer and must, in turn, explicitly pass control-flow to a successor block. Figure 3.2 illustrates the four possible block

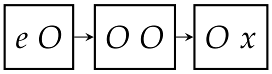
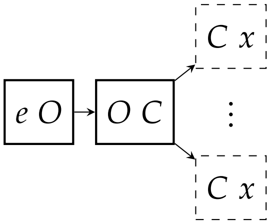
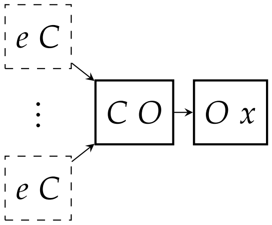
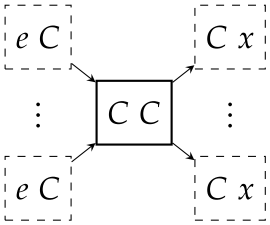
Shape	Example Graph	Example Statement
$O\ O$ (“open/open”)		Assignments.
$O\ C$ (“open/closed”)		Conditionals, jumps.
$C\ O$ (“closed/open”)		Function entry points, alternatives.
$C\ C$ (“closed/closed”)		Function bodies.

Figure 3.2: This table shows the four possible block shapes. Each row gives example statements and a representative CFG using a block of the given shape. Dashed lines indicate optional blocks. Solid lines show required blocks.

shapes, with representative examples.

Figure 3.3 gives Haskell declarations that can represent the AST for `example`. We use GHC's GADT syntax (The GHC Team, 2011a, Section 7.4.7) to specify the value of the *e* and *x* (“entry” and “exit”) types for each constructor, reflecting the control-flow of the represented statement. The *CExpr* and *Var* types do not affect control-flow in our subset, so we do not annotate them like *CStmt*. Hoopl defines the *Label* type; we use it to define the successors and predecessors of closed blocks.

```
data CStmt e x where
  Entry  :: Label → CStmt C O
  Assign :: Var → CExpr → CStmt O O
  Call   :: Var → [CExpr] → CStmt O O
  Return :: CStmt O C
data CExpr =
  Const Int
   | Add CExpr CExpr
   | Var Var
   | String String
type Var = String
```

Figure 3.3: Haskell data declarations for representing the AST of `example`.

The *Entry* value represents a function entry point; we give it type *C O* because control-flow can only explicitly enter a function through a call. The *Return* constructor creates a value with the type *CStmt O C*, meaning control-flow will not implicitly pass from the statement but rather explicitly transfer to another block (i.e., the caller of the function). The *Assign* constructor's type, *CStmt O O*, indicates that control-flow *will* fall-through, reflecting the behavior of the assignment statement.

The *Call* statement's type could be *O C* to reflect that control-flow implicitly enters the statement from its predecessor and then transfers explicitly to another block. However, we can think of this as an "external call" to a block defined outside the program. In this way *Call* acts like an assignment — control-flow implicitly passes through the function call to the next statement. Therefore, we give *Call* the type *O O*.

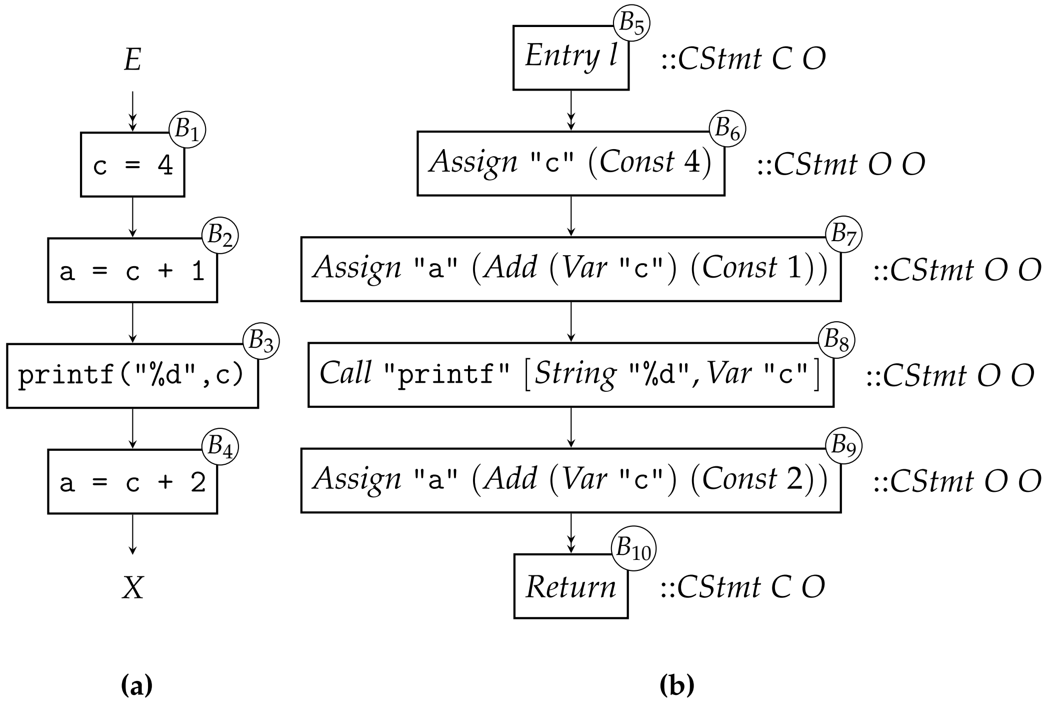


Figure 3.4: Our example function as a control-flow graph. Part 3.1 (a) uses C syntax for each statement. Part 3.1 (b) uses the AST given in Figure 3.3.

Figure 3.4 shows a CFG for `example`. Part (a) shows the program with C syntax. Part (b) uses the AST just given. Each block in Part (a) corresponds to the adjacent block in Part (b). For example, Block B_1 ("`c = 4`") corresponds to Block B_6 ("*Assign* `"c"` (*Const* 4)"). Also notice that the entry and exit points (*E* and *X*, respectively) in Part (a) do not appear explicitly in our program text, but they must be represented in the CFG.

Each block in Figure 3.4 (b) shows the type associated with its value. For example, the type of Block B_6, *CStmt O O*, shows that control-flow falls-through the statement. However, the type of B_5, *CStmt C O*, shows that control-flow must explicitly transfer to the block (in this case, through a function call). The type *CStmt O C* on B_{10} shows the opposite — control-flow does not implicitly exit the block; instead, control-flow explicitly returns to the caller of the function.

```
mkFirst   :: n C O → Graph n C O
mkMiddle  :: n O O → Graph n O O
mkLast    :: n O C → Graph n O C
(<*>)     :: Graph n e O → Graph n O x → Graph n e x
(|*><*|)  :: Graph n e C → Graph n C x → Graph n e x
```

Figure 3.5: Primitives provided by hoopl for constructing *Graph* values, representing cfgs.

Hoopl defines the type *Graph* for representing cfgs. Figure 3.5 shows the five primitive functions that hoopl provides for client programs to use for constructing cfgs. The *n* type in each primitive's signature represents the ast defined by the client program (*CStmt* in our example).

The *mkFirst*, *mkMiddle*, and *mkLast* functions turn a single block into a graph of one block with the same shape. The (<*>) operator, pronounced "concat," connects an "open on exit" (*e O*) graph to an "open on entry" (*O x*) graph. The resulting graph's shape, *e x*, combines the entry shape of the first argument and the exit shape of the second. Necessarily, the graph represented by the first argument becomes the predecessor of the graph represented by the second argument. For example, if *n1* has type *CStmt C O* and *n2* has type *CStmt O O*, then *n1* <*> *n2* would have type *CStmt C O* and *n1* will be the unique predecessor to *n2* in *n1* <*> *n2*. Hoopl combines smaller graphs into larger graphs using the (|*><*|) operator (pronounced "append"). Unlike (<*>), this operator does not imply any control-flow between its arguments.

```
example   :: Label → Graph CStmt C C
example l = mkFirst (Entry l) <∗>
  mkMiddle (Assign "c" (Const 4)) <∗>
  mkMiddle (Assign "a" (Add (Var "c") (Const 1))) <∗>
  mkMiddle (Call "printf" [String "%d", Var "c"]) <∗>
  mkMiddle (Assign "a" (Add (Var "c") (Const 2))) <∗>
  mkLast Return
```

Figure 3.6: A definition that creates a CFG for `example`, using the AST from Figure 3.3 and the functions shown in Figure 3.5.

Returning to our example, we can construct the CFG from Figure 3.4 (b) using the code in Figure 3.6. The *l* parameter (with type *Label*) defines the entry point for this block. Each statement in the block is mapped to a graph by applying *mkFirst*, *mkMiddle*, or *mkLast* as appropriate. We concatenate the graphs using the (<∗>) operator, forming one large graph with type *CStmt C C*. This construction exactly represents the CFG in Figure 3.4 (b).

The (<∗>) operator defines control-flow within a basic block, and the (|∗><∗|) operator combines unconnected blocks into a larger graph. Hoopl defines the *NonLocal* class to bridge the gap between these two operators:[1]

```
class NonLocal n where
  entryLabel :: n C x → Label
  successors :: n e C → [Label]
```

Hoopl defines the *Label* type, which the client program uses for two purposes:

[1] The (|∗><∗|) and (<∗>) operators in Figure 3.5 specify a *NonLocal* constraint on *n*, which we have hidden to simplify the presentation.

uniquely identifying each block and specifying the explicit successors of a given block. Hoopl use *entryLabel* method to find the entry point for a given block. The $n\ C\ x$ type of its argument ensures that *entryLabel* can only be applied to "closed on entry" nodes: precisely those nodes that can be the target of an explicit control-flow transfer. Similarly, hoopl uses *successors* to determine the explicit successors of a "closed on exit" block.

The client program must define an instance of *NonLocal* for its AST. Hoopl will use that instance to recover potential control-flow between basic blocks in the CFG. Therefore, as seen in Figure 3.3, the AST must store the label of a "closed on entry" node and the labels of successors for a "closed on exit" node in the AST itself.

We define the following instance of *NonLocal* for *CStmt*:

```
instance NonLocal CStmt where
  entryLabel (Entry l) = l
  successors Return    = []
```

We define *entryLabel* for *Entry*, the only "closed on entry" constructor for *CStmt*. Similarly, we just define *successors* for *Return*, the only "closed on exit" *CStmt* value. However, we do not specify the destination of a *Return* so *successors* always returns an empty list.

3.3 Facts, Meet Operators and Lattices

The dataflow algorithm, as given for the forwards case in Figure 2.10 on Page 23, iteratively computes output *facts* for each block in the CFG until reaching a fixed point. Input facts correspond to the $in(B)$ set for each block; output facts correspond to the $out(B)$ set for the block.[2] The first iteration uses some initial value

[2] A backwards analysis reverses this correspondence.

for each *in*(B) and *out*(B) set. Each subsequent iteration uses a *meet operator* to combine *out*(B) sets from the predecessors of each block into an *in*(B) set for that block. Together, the set of possible facts and the meet operator must form a *lattice*.

Hoopl provides the *DataflowLattice* type (shown in Figure 3.7), which defines the following fields: *fact_name*, used for documentation; *fact_bot*, for specifying initial facts; and *fact_join*, for the implementation of the analysis' meet operator.[3]

```
data DataflowLattice a = DataflowLattice {
  fact_name :: String,
  fact_bot  :: a,
  fact_join :: Label → OldFact a → NewFact a → (ChangeFlag, a) }
newtype OldFact a  = OldFact a
newtype NewFact a = NewFact a
data ChangeFlag = NoChange | SomeChange
```

Figure 3.7: *DataflowLattice* and associated types defined by hoopl for representing and combining facts.

The meet operator, *fact_join*, takes three arguments and returns a pair consisting of a value and a *ChangeFlag*. The arguments represent possibly differing output facts; the result represents the meet of those facts. Hoopl determines that a fixed point has been reached when *fact_join* returns *NoChange* for all blocks in the cfg.[4] The client program must ensure that the meet defines a finite-height lattice;

[3] The hoopl authors choose to document their library in terms of *joins*. We follow (Aho et al., 2006) and use the meet.

[4] Hoopl uses this strategy for efficiency: if the client does not specify when facts change, hoopl would need to do many comparisons on each iteration to determine if a fixed point had been reached.

otherwise, the analysis may not terminate.

As stated previously, dead-code elimination uses *liveness* analysis to find dead code. A variable is live at a given point if it will be used at a later execution point; otherwise the variable is dead. Liveness analysis is implemented as a backwards dataflow analysis. In a backwards analysis, $out(B)$ is the set of input facts to block B; $in(B)$ is the set of output facts. All live variables from B's successors may be live in B; therefore, we implement our meet operator as *set union*: to compute $out(B)$ for block B, we take the union of all the *in* sets of B's successors.

We define the set Vars as the set of all declared variables in the program. For each block B, our analysis computes the set of variables that are live at the beginning of each block, $in(B)$, using the transfer function defined in Section 3.4) and $out(B)$, the block's input set. Both $in(B)$ and $out(B)$ are subsets of Vars. We set $in(B)$ and $out(B)$ to the empty set when analysis begins.

Figure 3.8 shows Haskell code that implements the definitions of the meet operator and facts just given. The type *Vars* corresponds to Vars. The definition of *meet* corresponds to set union. If *old* does not equal *new* we return *SomeChange* and the union of the two sets (the *changeIf* function maps *Bool* values to *ChangeFlag* values). The *lattice* definition puts all the pieces together into a *DataflowLattice* value. Notice we set *fact_bot* to *Set.empty*, the initial value for all $in(B)$ and $out(B)$ sets.

3.4 Direction & Transfer Functions

The dataflow algorithm specifies two sets of facts for every block in the cfg: $in(B)$ and $out(B)$. $in(B)$ represents facts known when control-flow enters the block; $out(B)$ those facts known when control-flow leaves the block. The *transfer function*

```
type Vars = Set Var
meet :: Label → OldFact Vars → NewFact Vars → (ChangeFlag, Vars)
meet _ (OldFact old) (NewFact new) = (changeIf (old /= new), old `union` new)
lattice :: DataflowLattice Vars
lattice = DataflowLattice {
  fact_name = "Liveness"
  ,fact_bot  = Set.empty
  ,fact_join = meet}
```

Figure 3.8: Haskell definitions implementing fact and meet definitions for our liveness analysis.

computes output facts for each block in the CFG, using the contents of the block and its input facts. A forwards analysis uses *in*(B) as the input facts and computes *out*(B); A backwards analysis does the opposite, computing *in*(B) from *out*(B) and the contents of the block.

Hoopl defines the *FwdTransfer* and *BwdTransfer* types, shown in Figure 3.9, to represent forwards and backwards transfer functions. The n parameter represents the block's contents (i.e., the AST of the program under analysis). The f parameter represents the facts computed. Hoopl does not export the constructors for *FwdTransfer* or *BwdTransfer*; instead, clients use the *mkFTransfer* and *mkBTransfer* functions, whose signatures are also shown in Figure 3.9.

Hoopl requires that we parameterize our AST (i.e., the n type) using the O and C types from Section 3.2. A standard Haskell function cannot be applied to values with different types (e.g., *Assign* has type *CStmt O O*, but *Entry* has type *CStmt C O*). Therefore, to pattern-match on all constructors, *mkFTransfer* and

```
newtype FwdTransfer n f
newtype BwdTransfer n f

mkFTransfer :: (forall e x . n e x → f → Fact x f) → FwdTransfer n f
mkBTransfer :: (forall e x . n e x → Fact x f → f) → BwdTransfer n f
```

Figure 3.9: Hoopl's *FwdTransfer* and *BwdTransfer* types. They can be constructed with the functions *mkFTransfer* and *mkBTransfer*.

mkBTransfer require that the transfer function given be defined with a *higher-rank* type (The GHC Team, 2011a, Section 7.8.5). This allows client programs to write one transfer function that can match on all constructors in the AST.

Notice that *mkFTransfer* takes a transfer function that produces a *Fact x f* value. Hoopl defines *Fact x f* as an *indexed type family* (The GHC Team, 2011a, Section 7.2.2), where the meaning of *Fact x f* depends on the type of *x*. When *x* is *C*, then *Fact x f* is a synonym for *FactBase f* (another hoopl type), which is a dictionary of facts indexed by *Labels*. When *x* is *O*, *Fact x f* is just a synonym for *f* (i.e., a plain fact). The definition of *Fact x f* extends the dataflow algorithm slightly by allowing the transfer function to produce different facts for each successor node.

In the case of a backwards analysis, *mkBTransfer* specifies that the transfer function *receive* an argument of type *Fact x f*, and that it always produce a plain fact. When a node is closed on exit, the transfer function receives a dictionary of all facts (indexed by label) from the successors of the node. This definition also extends the dataflow algorithm slightly because it does not force the transfer function to take the meet of its input facts.

Figure 3.10 shows the implementation of the transfer function for our example. The subsidiary definition, *transfer*, computes facts for each constructor in *CStmt*:

transfer (*Entry* _) *f* This statement indicates the entry point of the function. Our AST does not represent parameters or globals; therefore, any variables live at this point were used but never assigned. Another analysis could use this fact to warn the programmer that an uninitialized variable was used. For our purposes, *transfer* just returns *f*, the set of facts so far.

transfer (*Assign var expr*) *f* In this case, *f* represents the set of live variables found so far. We first remove *var* from *f*, as assignment eliminates live variables. The auxiliary function *uses* computes the live variables in *expr*; we add those variables to our updated *f* and return the result.

transfer (*Call* _ *exprs*) *f* This case resembles *Assign*, except that we do not remove any variables from *f*. We add all variables used in any of the *exprs* given to the live set.

transfer Return f No variables (in our AST) will be live after the procedure returns. Therefore, nothing is live at this point, and we return the empty set.

3.5 Iteration & Fixed Points

The dataflow algorithm iterates over a program's CFG until the facts for each block reach a fixed point. Hoopl uses the meet operator (the *fact_join* field of the *DataflowLattice* type) given by the client to determine when the analysis should terminate.

```
liveness :: BwdTransfer CStmt Vars
liveness = mkBTransfer transfer
  where
    transfer :: forall e x . CStmt e x → Fact x Vars → Vars
    transfer (Entry _) f = f
    transfer (Assign var expr) facts = (var ‘delete‘ facts) ‘union‘ (uses expr)
    transfer (Call _ exprs) facts = facts ‘union‘ unions (map uses exprs)
    transfer Return _ = Set.empty
    uses :: CExpr → Set Var
    uses (Add e1 e2) = uses e1 ‘union‘ uses e2
    uses (Var v) = singleton v
    uses _ = Set.empty
```

Figure 3.10: The transfer function implementing liveness analysis.

Hoopl associates each block in the cfg with a *Label*. On each iteration, and at each label, hoopl computes the meet of facts from the prior iteration with facts from the current iteration. Recall that *fact_join* returns a *ChangeFlag*, as well as new facts. Therefore, if any application of *fact_join* results in *SomeChange*, hoopl continues to iterate. Otherwise, the analysis terminates.

3.6 Interleaved Analysis & Rewriting

Kildall's formulation of the dataflow algorithm (Kildall, 1973) does not give a general method for transforming cfgs based on the results of the analysis performed. He assumed that the cfg would be transformed after each analysis; he did not address the issue of determining when an analysis should be performed again (possibly leading to further rewrites). Kildall also did not address the question of composing multiple analyses; instead, each analysis is assumed to be applied one at a time, in no particular order.

Lerner, Grove and Chambers (2002) developed a variation of the dataflow algorithm that addresses these concerns. Kildall's dataflow algorithm computes facts over a static CFG; Lerner and colleagues' algorithm repeatedly transforms the CFG *during* analysis. After transforming some or all of the CFG, their algorithm re-computes facts for the new CFG; the transformation and analysis of the CFG continues until reaching a fixed point.

This algorithm does not produce better results than Kildall's, in the sense defined by Section 2.5 (Page 24). However, as Lerner and colleagues describe, their algorithm removes the need to combine individual dataflow analyses manually. Instead, each dataflow analysis can be implemented separately; their algorithm composes those separate pieces automatically. Hoopl implements a version of the interleaved analysis and rewriting algorithm just described.

3.7 Rewriting with Hoopl

Figure 3.11 shows the two types HOOPL provides for rewriting, *FwdRewrite* and *BwdRewrite*. These types correspond to the *FwdTransfer* and *BwdTransfer* types; HOOPL requires that a *FwdTransfer* be paired with a *FwdRewrite*, and a *BwdTransfer* with a *BwdRewrite*. Client programs use the *mkFRewrite* and *mkBRewrite* functions to create *FwdRewrite* and *BwdRewrite* values. For the same reason as the transfer function, rewrite functions must be defined with a higher-rank type.

The argument to *mkFRewrite* (and *mkBRewrite*) gives the signature for rewrite functions. A rewrite function receives the node to rewrite as its first argument. The facts computed for that node are given in the second argument. Like the backwards transfer function, a backwards rewriter receives a dictionary of facts, indexed by labels, if the node is closed on exit; otherwise, the rewriter receives a plain fact. A forwards rewriter always receives a plain fact.

The rewrite function returns a monadic *Maybe* (*Graph n e x*) value. The monadic portion relates to optimization fuel, a concept described in Section 3.7.1. The *Maybe* portion indicates if the rewriter wants to change the node given in any way. *Nothing* means no change to the node. A *Just* value causes hoopl to replace the current block with a *Graph n e x* value. Returning a *Graph* value allows the rewriter to replace a single node with many nodes, but the graph returned must have the same *e x* type (i.e., shape) as the input node.

Rewriters can delete *O O* nodes by returning *Just emptyGraph*. The shape type prevents *C O* and *O C* nodes from being deleted. To see why, consider the shape of each node and its successor (or predecessor, in the second case). A *C O* node necessarily precedes an *O x* node. If the *C O* node were deleted, the *O x* node cannot replace it. A *C O* node can have zero or more predecessors; a *O x* node can only have one. The predecessors to the *C O* node cannot become the predecessors to the *O x* node; therefore, deleting a *C O* node is not possible. A similar argument holds for *O C* nodes.

Figure 3.12 shows *eliminate*, the rewrite function for our example optimization. We define the local function *rewrite* by cases for each constructor in *CStmt*. All cases except *Assign* return *Nothing*, leaving the cfg unchanged. If the test *not* (*var* ‘*member*‘ *live*) in the *Assign* case succeeds, *rewrite* removes the assignment by returning *Just emptyGraph*. Otherwise, the assignment remains.

```
newtype FwdRewrite m n f
newtype BwdRewrite m n f

mkFRewrite :: FuelMonad m ⇒
  (forall e x . n e x → f → m (Maybe (Graph n e x)))
   → FwdRewrite m n f
mkBRewrite :: FuelMonad m ⇒
  (forall e x . n e x → Fact x f → m (Maybe (Graph n e x)))
   → BwdRewrite m n f
```

Figure 3.11: The *FwdRewrite* and *BwdRewrite* types provided by hoopl, as well as the functions used to construct them, *mkBRewrite* and *mkFRewrite*.

3.7.1 Optimization Fuel

Hoopl implements "optimization fuel," originally described by Whalley (1994), as an aid in debugging optimizations. Each rewrite costs one "unit" of fuel. If fuel runs out, hoopl stops iterating. This allows the programmer to debug faulty optimizations by decreasing the fuel supply in a classic divide-and-conquer approach. The *FuelMonad* constraint ensures hoopl can manage fuel during rewriting. Normally, the client program does not worry about fuel.

3.8 Executing an Optimization

Figure 3.13 shows hoopl's *BwdPass* and *FwdPass* types. The figure also shows the signatures for *analyzeAndRewriteBwd* and *analyzeAndRewriteFwd*, the hoopl functions that the client program uses to apply a given analysis and transformation. As the names suggest, one pair applies to backwards dataflow-analyses and the

```
eliminate :: FuelMonad m ⇒ BwdRewrite m CStmt Vars
eliminate = mkBRewrite rewrite
  where
    rewrite :: FuelMonad m ⇒ forall e x . CStmt e x
       → Fact x Vars → m (Maybe (Graph CStmt e x))
    rewrite (Entry _) _ = return Nothing
    rewrite (Assign var exprs) live = return $
      if not (var ‘member‘ live)
        then Just emptyGraph
        else Nothing
    rewrite (Call _ _) _ = return Nothing
    rewrite Return _ = return Nothing
```

Figure 3.12: The rewrite function for our dead-code elimination optimization. *Assign* statements are deleted when they assign to a dead variable. In all other cases the CFG remains unchanged.

other to forwards analyses. We will only discuss the backwards case here.

The *BwdPass* type packages a lattice definition, transfer function, and rewrite function into one structure. The *analyzeAndRewriteBwd* function takes a number of arguments and must be run inside a HOOPL-specified monad. We address each argument in turn.

(*CheckpointMonad m*, *NonLocal n*, *LabelsPtr entries*) **— Line 15** These constraints reflect several HOOPL requirements:

- *CheckpointMonad* – This class provides methods that allow HOOPL to rollback monadic changes to the CFG, providing support for HOOPL's implementation of Lerner and colleague's technique.
- *NonLocal* – This class allows HOOPL to traverse the CFG.

```
data FwdPass m n f = FwdPass {
  fp_lattice :: DataflowLattice f
  ,fp_transfer :: FwdTransfer n f
  ,fp_rewrite :: FwdRewrite m n}

data BwdPass m n f = BwdPass {
  bp_lattice :: DataflowLattice f
  ,bp_transfer :: BwdTransfer n f
  ,bp_rewrite :: BwdRewrite m n f}

analyzeAndRewriteFwd :: (CheckpointMonad m, NonLocal n, LabelsPtr entries) ⇒
  FwdPass m n f
   → MaybeC e entries
   → Graph n e x
   → Fact e f
   → m (Graph n e x, FactBase f, MaybeO x f)

analyzeAndRewriteBwd :: (CheckpointMonad m, NonLocal n, LabelsPtr entries) ⇒
  BwdPass m n f
   → MaybeC e entries
   → Graph n e x
   → Fact x f
   → m (Graph n e x, FactBase f, MaybeO e f)
```

Figure 3.13: Hoopl's types and functions used to execute backwards and forwards analysis and transformation. *BwdPass* and *FwdPass* package the client program's definition of lattice, transfer function, and rewrite function. Except for direction, *analyzeAndRewriteFwd* and *analyzeAndRewriteBwd* behave similarly; they execute the optimization defined by the client program.

- *LabelsPtr* – This class gives hoopl the means to find external entry points to the cfg.

BwdPass m n f **— Line 16** This argument packages the client's definitions of the lattice, transfer function, and rewrite function for this particular analysis.

MaybeC e entries **— Line 17** This gives all the entry points to the program, which may not always be all the *Labels* in the cfg.

Graph n e x **— Line 18** This argument holds the cfg to be optimized. In practice, *e x* is always *C C*.

Fact x f **— Line 19** This argument gives the initial input facts for all nodes in the graph.

Figure 3.14 shows *deadCode*, which puts all the pieces of our example optimization together and applies them to a given program. The type, *Graph CStmt C C* → *Graph CStmt C C*, shows that *deadCode* modifies a cfg composed of *CStmt* values.

The *opt* definition implements our analysis and transformation. Our analysis must run in a monadic context that is an instance of *CheckpointMonad* and *UniqueMonad* (a class that controls the creation of new *Label* values — allowing rewriters to create new *C x* nodes). The *CheckingFuelMonad* and *SimpleUniqueMonad* types in the signature of *opt* are the hoopl-provided implementations of *CheckpointMonad* and *UniqueMonad*.

The first argument to *analyzedAndRewriteBwd*, *pass*, packages up the lattice definition, transfer function, and rewrite function previously discussed. The second argument, (*JustC entryPoints*), gives all entry points for the program. The third argument is the program we are optimizing. Finally, the input facts (an empty set) are given in the fourth argument.

analyzeAndRewriteBwd returns a transformed program, the final facts computed, and any facts that should propagate "out" of the CFG. We capture the transformed program in *program′* and return it.

In *deadCode*, we use (*runWithFuel infiniteFuel*) and *runSimpleUniqueMonad* (all provided by HOOPL) to execute the monadic program returned by *opt* and ultimately, we return the transformed program.

```
deadCode :: Graph CStmt C C → Graph CStmt C C
deadCode program = runSimpleUniqueMonad $ runWithFuel infiniteFuel $ opt
  where
    opt :: CheckingFuelMonad SimpleUniqueMonad (Graph CStmt C C)
    opt = do
      (program′, _, _) ← analyzeAndRewriteBwd pass (JustC entryPoints) program
                                              facts
      return program′
    pass = BwdPass {bp_lattice = lattice, bp_transfer = liveness
                   , bp_rewrite = eliminate}
    entryPoints = case program of
      (GMany _ blocks _) → map (entry . blockToNodeList′) (mapElems blocks)
    entry :: (MaybeC e (CStmt C O), [CStmt O O], MaybeC x (CStmt O C))
       → Label
    entry (JustC (Entry l), _, _) = l
    facts :: FactBase Vars
    facts = mkFactBase lattice (zip entryPoints (repeat Set.empty))
```

Figure 3.14: *deadCode* applies the optimization developed so far to a particular program.

3.9 Summary

This chapter gave an introduction to the essential features of the hoopl library. Hoopl implements the generic portions of the dataflow algorithm; in particular, it determines when facts reach a fixed point. Hoopl's implementation of the dataflow algorithm interleaves analysis and rewriting, a technique originally described by Lerner and colleagues (Lerner, Grove, and Chambers, 2002). Hoopl requires that the client program define the facts to analyze, a transfer function, a rewriting function, and a meet operator (which, in turn, defines a lattice for the facts given). The complete source code for the dead-code elimination optimization shown in this chapter can be downloaded from the url given in the Appendix.

Chapter 4

A Monadic Intermediate Language

Most compilers do not generate executable machine code directly from a program source file. Rather, the compiler transforms the program into a number of different *intermediate representations*, where each representation exposes different details about the implementation of the program. Although not intended to be used directly, many of these intermediate representations are languages in their own right.

A broad range of intermediate languages have been described for both imperative and functional languages. *Three-address code* (described in standard textbooks such as Aho et al., 2006), a language normally used to represent imperative languages, emphasizes simplicity by requiring that all operations specify, at most, two operands and one destination. Three-address code aids in optimizing the use of registers, a scarce resource on most processors. Administrative-Normal Form (ANF), first described by Flanagan et al. (1993) and intended for functional languages, requires that all intermediate values be named. ANF's inventors proposed it as a replacement for continuation-passing style (CPS), a widely-employed intermediate representation most completely described by Appel (1992). The simple structure of both CPS and ANF programs eases their translation to executable assembly-like languages.

Monadic programming, popularized by Wadler (1990), but first described by

Moggi (1991), separates expressions into those that have side-effects and those that do not. A side-effecting expression is considered a computation, or program, that must execute to produce a value. Running the program multiple times may produce different results.

In this chapter, we describe a *monadic intermediate language*, mil, that exploits monadic programming to segregate side-effecting operations (such as allocations) from pure operations, such as case discrimination or jumps. More specifically, mil supports functional language features, but also follows the form of three-address code. Mil directly supports function application and abstraction. All intermediate values are named. Mil specifies evaluation order and separates stateful computation using a monadic programming style. Mil's syntax enforces basic-block structure on programs, making them ideal for dataflow analysis.

In order to write programs that we can translate to mil, we use a simple variant of the λ-calculus, called λ_C, that supports a monadic programming style, case discrimination, local definitions, and other features of a high-level, purely functional language. We describe λ_C in Section 4.1. The motivation and roots of mil are given in Section 4.2. Mil's complete syntax follows in Section 4.3. Section 4.4 discusses mil's treatment of allocation as a side-effect. Section 4.5 shows how mil treats monadic programs as suspended computations. Section 4.6 highlights some of the subtler points in our translation strategy from λ_C to mil. We sketch how mil programs can be evaluated in Section 4.7. Section 4.8 shows how hoopl influenced the ast that we use to represent mil programs. In Section 4.10 we discuss the mlj and sml.net compilers, both of which also used a monadic intermediate language. We conclude in Section 4.11.

4.1 Source Language: λ_C

Our source language, λ_C (pronounced "lambda-case"), derives from the λ-calculus, with elements borrowed from the Haskell language's use of **do** notation to represent monadic programs. The HASP group developed λ_C as an intermediate representation for the Habit programming language (2010). The Habit "Compilation Strategy" report (2010) gives full details on λ_C.[1] The report describes several different intermediate languages; λ_C corresponds to "Normalized MPEG."

Figure 4.1 gives the full syntax of λ_C. In the figure, x, x_1, etc. represent simple variables, while t, t_1, etc. represent arbitrary terms. All *def* terms are global to the program in which they are defined. Definitions with zero parameters are values and cannot be recursive; definitions with more than one parameter are functions and can be recursive. Only variables can appear as arguments in definitions and case alternatives — the language does not support more sophisticated pattern-matching. While most elements should be recognizable from Haskell or the λ-calculus, we will explain the the *monadic bind* and *primitive* terms further.

Monadic Bind The monadic binding term, $\textbf{do}\ \{x \leftarrow t_1; t_2\}$, states that the result of running the monadic computation, t_1, will be bound to x and that t_2 will then be executed with x in scope. Note that x can only be a variable; λ_C does not support pattern-matching on the left-hand side of a bind. When t_2 is another monadic bind, we do not nest the **do** keyword: for brevity, we write $\textbf{do}\ \{x \leftarrow t_1; y \leftarrow t_2; t\_3\}$ rather than $\textbf{do}\ \{x \leftarrow t_1; \textbf{do}\ \{y \leftarrow t_2; t\_3\}\}$.

Primitives The primitive expression p^* refers to a primitive definition named

[1] For simplicity's sake, we ignore two features of λ_C in this work: patterns and guards. Details on those elements can be found in the aforementioned report.

$$
\begin{array}{rlr}
\mathit{def} := & f\ x_1\ \ldots\ x_n = \mathit{term}, n \geqslant 0 & \mathit{Definition} \\
\\
\mathit{term} := & x, x_1, x_2, \ldots & \mathit{Variables} \\
| & \lambda x.\ t_1 & \mathit{Abstraction} \\
| & t_1\ t_2 & \mathit{Application} \\
| & \mathbf{case}\ t\ \mathbf{of} & \mathit{Case\ Discrimination} \\
 & \quad \mathit{alt}_1 & \\
 & \quad \ldots & \\
 & \quad \mathit{alt}_n & \\
| & \mathbf{do}\ \{x \leftarrow t_1; t_2\} & \mathit{Monadic\ Bind} \\
| & \mathbf{let}\ \mathit{def}_1 & \mathit{Let} \\
 & \qquad \ldots & \\
 & \qquad \mathit{def}_n & \\
 & \mathbf{in}\ t & \\
| & p^* & \mathit{Primitive} \\
| & C\ x_1 \ldots x_n, n \geqslant 0 & \mathit{Allocate\ Data} \\
\\
\mathit{alt} := & C\ x_1\ \ldots\ x_n \rightarrow t, n \geqslant 0 & \mathit{Alternative}
\end{array}
$$

Figure 4.1: The syntax of λ_C. Variables are represented using x, x_1, etc. Terms are represented by t, t_1, etc. C represents the name of a given constructor.

p. Primitives refer to functionality that is not implemented in λ_C itself. In all other respects the primitive p^* is treated like any other function value.

4.2 MIL's Purpose

The design of an intermediate language typically exposes some specific implementation details while hiding others in order to support certain analysis and transformation goals. Exposing intermediate values gives us the chance to analyze and eliminate them. Hiding implementation details makes the job of writing those analyses and transformations simpler.

Mil's syntax and design borrow heavily from three-address code, as stated previously. Three-address code represents programs such that all operations specify, at most, two operands and a single destination. Three-address code hides details of memory management by assuming that arbitrarily many storage locations can be named and updated. For example, the expression:

$$\frac{(b * c + d)}{2},$$

could be expressed in three-address code as:

```
t_1 <- mul b c
t_2 <- add t_1 d
t_3 <- div t_2 2
```

where $\mathtt{t}_1$, $\mathtt{t}_2$ and $\mathtt{t}_3$ represent temporary storage locations and `mul`, `add`, and `div` represent the corresponding arithmetic operations.

Mil seeks to expose certain effects that are not normally represented in functional languages. As described by Wadler (1990), *monads* can be used distinguish *pure* and *impure* functions. A *pure* function has no side-effects: it will not in any way change the observable state of the machine. An *impure* function may change the machine's state in an observable way. Most functional languages treat data and closure allocation as pure operations. Mil uses monads to treat those allocations as impure operations.

Three-address code emphasizes assignments and low-level operations, features important to imperative languages. Mil emphasizes allocation, higher-order functions and side-effecting computations, features important to functional languages. Though the operations supported by MIL differ from traditional three-address code, the intention remains the same: hide some details while exposing those that we care about.

4.3 MIL Syntax

Figure 4.2 gives the syntax for MIL. Where the term $\mathtt{v}_1$, etc. appears, only simple variables are allowed. This includes most terms in the language, staying true to the design of three-address code. Bold terms such as **b** and **k** represent labeled locations in the MIL program. `C` represents the name of a data constructor. Bold text such as **case** and **invoke** represent MIL keywords. A MIL program consists of a number of labeled blocks.

Closure-capturing blocks specify an environment, $\{\mathtt{v}_1, \ldots, \mathtt{v}_n\}$, and an argument, `v`. Closure-capturing blocks only execute when initiated by an *enter* expression of the form `f @ x`. In `f @ x`, `f` refers to a closure that will point to a closure-capturing block named **k**. The environment declared for **k** corresponds to the environment captured by the closure. The argument `x` becomes the argument `v` declared in the closure-capturing block. We chose to allow only a single tail expression in the body of a closure-capturing block in order to simplify analysis of their behavior.

Basic blocks consist of a sequence of monadic binding statements that execute in order without any intra-block jumps or conditional branches. The parameters to the block, $(\mathtt{v}_1, \ldots, \mathtt{v}_n)$, are the only variables in scope at the start of the block. The name of the block (**b**) is global to the program. Each basic block ends with either a **case** statement or a *tail*.

Each *monadic binding* statement assigns the result of the *tail* on the right-hand side of the statement to the variable on the left. If a variable is bound more than once, later bindings will shadow previous bindings.

The *case discrimination* statement examines a discriminant and selects one

variable:=	$\mathtt{v_1,\ldots,v_n}$		*Variables*
block:=	$\mathbf{k}\,\{\mathtt{v_1,\ldots,v_n}\}\ \mathtt{v}$: *tail*		*Closure-Capturing Block*
|	$\mathbf{b}\,(\mathtt{v_1,\ldots,v_n})$: *bind₁* ... *bindₙ* *done*		*Basic Block*
bind:=	`v <-` *tail*		*Monadic Bind*
done:=	**case** `v` **of** *alt₁* ... *altₙ*		*Case Discrimination*
|	*tail*		
alt:=	$\mathtt{C\ v_1\ \ldots\ v_n}$ `->` $\mathbf{b}\,(\mathtt{v_1,\ldots,v_n})$		*Case Alternative*
tail:=	**return** `v`		*Return*
|	$\mathtt{v_1}$ `@` $\mathtt{v_2}$		*Enter*
|	$\mathbf{b}\,(\mathtt{v_1,\ldots,v_n})$		*Goto Block*
|	$\mathbf{p}^*(\mathtt{v_1,\ldots,v_n})$		*Goto Primitive*
|	$\mathbf{k}\,\{\mathtt{v_1,\ldots,v_n}\}$		*Allocate Closure*
|	$\mathbf{b}\,[\mathtt{v_1,\ldots,v_n}]$		*Allocate Monadic Thunk*
|	**invoke** `v`		*Execute Thunk*
|	$\mathtt{C\ v_1\ \ldots\ v_n}$		*Allocate Data*

Figure 4.2: Complete syntax for MIL.

alternative based on the value found. The discriminant is always a simple variable, not an expression. Each *case alternative* specifies a *constructor* and variables for each value held by the constructor. No "default" alternative exists — all possible constructors must be specified. Again, to simplify later analysis, we chose that alternatives must always jump immediately to a block — they do not allow any other expression.

Tail terms represent potentially effectful computations and always appear on the right-hand side of a bind statement, at the end of a basic block, or as the body of a closure-capturing block. **`return`** takes a variable (*not* an expression) and produces a computation that returns that value with no side-effects. The "enter" operator, `@`, implements function application, "entering" the closure represented by its left-hand side with the argument on its right-hand side. The "*goto block*" and "*goto primitive*" expressions implement labeled jumps with arguments. In the first case, `b` represents a labeled block elsewhere in the program. The primitive term, $\mathtt{p}^*$, also represents a labeled location, except that the body of the primitive is not implemented in MIL. Otherwise, *goto block* and *goto primitive* are treated the same.

Closure allocation, written as $\mathtt{k}\,\{\mathtt{v}_1,\ldots,\mathtt{v}_\mathtt{n}\}$, creates a closure pointing to the closure-capturing block `k`, capturing the variables $\mathtt{v}_1,\ldots,\mathtt{v}_\mathtt{n}$. No other type of block can be referenced in a closure — `k` always refers to a closure-capturing block.

The term $\mathtt{b}\,[\mathtt{v}_1,\ldots,\mathtt{v}_\mathtt{n}]$ allocates a *monadic thunk*. The thunk stores a reference to block `b` and captures the variables $\mathtt{v}_1,\ldots,\mathtt{v}_\mathtt{n}$. Unlike closures, thunks do not store references to closure-capturing blocks. Instead, `b` always refers to a basic block or primitive. The **`invoke`** operator executes the *monadic thunk* referred to by its argument. We describe thunks in Section 4.5.

The constructor expression $\mathtt{C}\ \mathtt{v}_1\ \ldots\ \mathtt{v}_\mathtt{n}$ creates a data value with the given tag, `C`, and the values $\mathtt{v}_1,\ldots,\mathtt{v}_\mathtt{n}$ in the corresponding fields.

MIL Example: *compose*

Consider the definition of *compose* given in Figure 4.3 (a) and the corresponding MIL program in Part (b). The three closure-capturing blocks, **`k1`**, **`k2`** and **`k3`** correspond to the three λ values $(\lambda f.\ \ldots)$, $(\lambda g.\ \ldots)$, and $(\lambda x.\ \ldots)$. Each block, except **`k3`**, captures a single argument and returns a new closure holding previously captured values plus the new argument. **`k3`** executes when all arguments are captured and immediately jumps to **`compose`**, the block implementing the body of *compose*.

The basic block defined on Line 4 gives the name of the block (**`compose`**) and its parameters (`f`, `g`, and `x`). Line 5 applies `g` to `x` and assigns the result to `t1`. The "enter" operator (`@`), implements function application. [2]

The "bind" operator (`<-`) assigns the result of the operation on its right-hand side to the location on the left. All expressions that could have a side-effect appear on the right-hand side of a bind operator in MIL; in this case, `g @ x` may allocate memory when evaluated.

Line 6 applies `f` to `t1` and assigns the result to `t2`. The last line returns `t2`. Thus, the **`compose`** block returns the value of $f\ (g\ x)$, just as in our original λ_C expression.

4.4 Allocation as a Side-Effect

Functional languages normally treat data allocation as a hidden operation in that the program cannot directly observe any effect from an allocation. Of course, in practice, allocation can cause effects, such as updating the heap or triggering a garbage collection. For example, using the definition of *compose* given in Figure 4.3 (a), consider the sequence of applications that occur when calculating *compose a b c* using call-by-value evaluation:

[2] So called because in the expression `g @ x`, we "enter" function `g` with the argument `x`.

$$compose \;=\; \lambda f.\ \lambda g.\ \lambda x.\ f\ (g\ x)$$

```
k1{} f: k2{f}
k2{f} g: k3{f, g}
k3{f, g} x: compose(f, g, x)

compose(f, g, x):
  t1  <- g @ x
  t2  <- f @ t1
  return t2
```

(a) **(b)**

Figure 4.3: Part (a) gives a λ_C definition of the composition function; (b) shows a fragment of the MIL program for *compose*.

$$\begin{aligned}
compose\ a\ b\ c \;&= (\lambda f.\ \lambda g.\ \lambda x.\ f\ (g\ x))\ a\ b\ c\\
&= (\lambda g.\ \lambda x.\ a\ (g\ x))\ b\ c\\
&= (\lambda x.\ a\ (b\ x))\ c\\
&= a\ (b\ c)
\end{aligned}$$

This notation hides an important detail: each function application potentially allocates a closure representing the function and its environment. For example, $(\lambda g.\ \ldots)\ b$ might allocate a closure referring to the function $(\lambda g.\ \lambda x.\ f\ (g\ x))$ and mapping the free variable f to the value of a.

Figure 4.4 shows *compose* rewritten in a monadic style that makes closure allocations explicit, and a program that uses the rewritten *compose* to evaluate *compose a b c*. The *closure* operation in Part 4.4 (a) is a monadic function that allocates memory and stores its arguments for retrieval later.[3] The first argument is the function that the closure points to. The second argument represents the environment that will be used when that function is evaluated.

[3] λ_C does not support pattern-matching on function arguments, but we use them in Part 4.4 (a) for clarity here.

```
k0 = do
  t ← closure k1 [ ]
  return t
k1 [ ] f = do
  t ← closure k2 [f]
  return t
k2 [f] g = do
  t ← closure compose [f,g]
  return t
compose [f,g] x = do
  t ← f (g x)
  return t
```

(a)

```
main a b c = do
  t0 ← k0
  t1 ← app t0 a
  t2 ← app t1 b
  t3 ← app t2 c
  return t3
```

(b)

Figure 4.4: Part 4.4 (a) shows a rewritten version of *compose* that makes closure allocation explicit; Part 4.4 (b) gives a program that evaluates *compose a b c*. Note that these programs produce monadic values rather than pure values.

Each λ in Figure 4.3 (a) becomes a k_n definition in Figure 4.4 (a). k_0 corresponds to the expression $\lambda f.\ \lambda g.\ \lambda x.\ f\ (g\ x)$, allocating a closure with an empty environment and pointing to k_1. k_1 corresponds to $\lambda g.\ \lambda x.\ f\ (g\ x)$, while k_2 corresponds to $\lambda x.\ f\ (g\ x)$. Each k_n definition (except k_0) takes two arguments. The first is a list of values representing the environment of the function and the second a new value representing the argument of the original λ expression. k_1 and k_2 each allocate and return a new closure. The closure stores a reference to the next k_{n+1} definition and a list of values representing the current environment. *compose* evaluates $f\ (g\ x)$; however, this expression also becomes monadic, because $f\ (g\ x)$ may cause an allocation.

Figure 4.4 (b) shows a program that evaluates *compose a b c* using our monadic version of *compose*. To evaluate each closure, we define a monadic function *app* that takes a closure and an argument. *app* evaluates the function referred to by the closure, using the environment stored in the closure and passing the additional argument supplied. The first line of the program in Figure 4.4 (b) allocates an initial closure that serves as the entry point for *compose*. Each line after gathers an additional argument and "adds" it to the closure represented. The final line returns the result of *compose a b c*. In our original expression, the potential side-effect caused by each application was not clear. In *main*, each line makes it very clear that a side-effect may occur.

Of course, in λ_C we do not really define the *closure* or *app* functions and closure allocation is not directly visible. Mil, however, treats allocation as an impure operation and makes it explicit. Figure 4.5 shows a complete mil program for $main = compose\ a\ b\ c$. `main` corresponds to the definition of *main* in Figure 4.4 (b). `k0` corresponds to the k_0 definition in Figure 4.4 (a) and acts as the entry point for *compose*. Notice that `k0` is not a closure-capturing block; `k0` just allocates the initial closure that will be used to evaluate *compose a b c*. The closure-capturing blocks `k1` and `k2` correspond to k_1 and k_2. The second argument to each *closure* operation corresponds to the variables captured by the closures allocated in the body of `k1` and `k2`; the list of values passed as an argument to k_1 and k_2 correspond to the environment defined for the `k1` and `k2` closure-capturing blocks. In Figure 4.4, k_2 creates a closure pointing to *compose*. Because closure-capturing blocks can only contain a single *tail* term, `k3` jumps immediately to `compose`, which implements the body of *compose*.

```
1  main(a, b, c):
2    t0 <- k0();
3    t1 <- t0 @ a
4    t2 <- t1 @ b
5    t3 <- t2 @ c
6    return t3

7  k0(): k1{}
8  k1{}f: k2{f}
9  k2{f}g: k3{f, g}
10 k3{f, g}x: compose(f, g, x)

11 compose(f, g, x): ... as in Figure 4.3 (b) ...
```

Figure 4.5: The MIL program which computes *main a b c* = *compose a b c*.

By examining `main` in Figure 4.5, we can see how MIL makes explicit the intermediate closures created while evaluating *compose a b c*. Line 2 executes the block `k0`, allocating a closure pointing to `k1` and assigning it to `t0`. On line 3, we apply `t0` to `a`; `k1` executes and creates a closure that points to `k2` and holds the value of `a`. We assign `k2{a}` to `t1`. On Line 4 we apply `t1` to the second argument, `b`. This executes `k2`, which expects to find one argument in its environment, just as we stored in `t1`. `k2` creates another closure, `k3{a,b}`, which we assign to `t2`. This closure points to `k3` and holds two variables in its environment. Finally, on line 5, we apply `t2` to the final argument, `c`. `k3` executes and immediately jumps to `compose` with our arguments. The result, assigned to `t3`, is returned on the last line of `main`.

4.5 Monadic Thunks

As described in Wadler's 1990 paper, a monadic value represents a *computation*. Where a pure value evaluates without side-effects, a monadic value represents a suspended computation that may cause side-effects when evaluated. Evaluating the computation multiple times may even produce different side-effects each time.

Consider the λ_C functions in Figure 4.6.[4] Neither takes any arguments and they ostensibly produce the same number. Of course, the value produced by the pure function in Part (a) differs markedly from that produced by the impure function in Part (b).

$$\mathit{num} = 1$$

(a)

$$\begin{aligned}&\mathit{printNum} = \textbf{do}\\ &\quad \mathit{print}\ 1\\ &\quad \mathit{return}\ 1\end{aligned}$$

(b)

Figure 4.6: Part 4.6 (a) shows a *pure* value. Part 4.6 (b) shows an *impure* value.

Intuitively, *num* returns 1, but *printNum* returns a *computation*. We call this computation a *monadic thunk*. Traditionally, thunks represent *suspended* computation. We use it in the same sense here in that *printNum* evaluates to a program that we can invoke; moreover, evaluating *printNum* alone (as in the expression **let** $x = \mathit{printNum}$) will *not* invoke the computation — *printNum* must be evaluated and then invoked before the computation will produce a result.

[4] Some syntactic liberties have been taken here.

To illustrate, consider the λ_C functions in Figure 4.7 (a).[5] The *echo* function prints its argument to the screen. The **do** keyword shows that *echo* is a monadic function. The *main* function uses a **let** statement to assign *m* the value *echo a*. Notice this does *not* evaluate *echo a*; instead, *m* is a thunk that points to the *echo* function and that captures the value of *x*.

```
echo a = do
  print a

main x = do
  let m = echo x
  m
  m
```

(a)

```
1 echo(a): print*(a)

2 main(x):
3   m <- echo[x]
4   _ <- invoke m
5   invoke m
```

(b)

Figure 4.7: Part (a) shows two monadic λ_C functions. The MIL blocks that create and use monadic thunks to execute *main* are shown in Part (b).

Part (b) shows the corresponding MIL code for *echo* and *main*. The `echo` block on Line 1 merely executes the primitive `print`*. The `main` block, however, shows how we allocate and invoke a thunk. Line 3 allocates the thunk referring to `echo` and capturing `x`. The thunk is bound to `m`. Lines 4 and 5 invoke the thunk causing `echo` to execute twice. Notice we do not allocate the thunk again — only one allocation occurs, but we run the `echo @ x` "program" twice.

4.6 Compiling λ_C to MIL

We created a simple compiler from λ_C to Mil in order to generate tests for the optimizations discussed later in this work. Our compiler implementation follows

[5] Again, some syntactic liberties are taken.

the style used by Kennedy (2007) and in most cases it does not differ much from numerous other compilers that translate the λ-calculus to a given intermediate form. However, we will highlight some nuances of our translation.[6]

Consider again the definition of *compose* in Figure 4.3 (a) on Page 61. Our compiler translates each λ, except the innermost (i.e., $\lambda x.\ f\ (g\ x)$), to a block that returns a closure. The innermost λ translates to a block that immediately jumps to an implementation of the body of *compose*. This gives the sequence of blocks shown in Figure 4.5 on Page 64 (excepting `main`, of course) and allows MIL to support partial application.

While general, this strategy produces code that does a lot of potentially unnecessary work. When fully applied, a function of n arguments will create $n-1$ closures, perform $n-1$ jumps between blocks, and potentially copy arguments between closures each time. Our uncurrying optimization (Chapter 5) can collapse this work to one closure, one jump, and no argument copying in many cases. Therefore, generating simple (and easy to analyze) code is a reasonable trade-off.

Monadic code presents other challenges. Monadic expressions do not directly produce a value; they produce a thunk to be evaluated later. Therefore, when the compiler first encounters a monadic expression, it generates a block that returns a thunk. The block pointed to by the thunk executes the monadic expression.

The code shown for *kleisli* in Figure 4.8 illustrates this strategy. The *kleisli* function in Part 4.8 (a) implements *monadic* composition, in which f and g produce monadic results. Part 4.8 (a) shows a MIL implementation of *kleisli*. Block `m205` on

[6] Those interested in the full compiler can download the source from the URL given in the Appendix.

Line 6 executes the body of *kleisli*. However, no blocks call `m205` directly. Instead, block `b204` on Line 5 returns a thunk that points to `m205` and captures all the arguments to `m205` (`g`, `f`, and `x`). `b204` only executes after all arguments for *kleisli* are collected; `m205` can only execute by invoking the thunk returned by `b204`.

```
kleisli f g x = do
  v ← g x
  f v
```

(a)

```
 1 kleisli (): k201 {}
 2 k201 {} f: k202 {f}
 3 k202 {f} g: k203 {g, f}
 4 k203 {g, f} x: b204 (g, x, f)
 5 b204 (g, x, f): m205 [g, x, f]

 6 m205 (g, x, f):
 7   v207 <- g @ x
 8   v1 <- invoke v207
 9   v206 <- f @ v1
10   invoke v206
```

(b)

Figure 4.8: Part (a) shows a λ_C implementation of the monadic composition function (sometimes called "Kleisli composition"). Part (b) shows a MIL implementation of the same function.

While the compiler follows the strategy above when it first encounters a monadic expression, it changes strategy when compiling binding statements that follow the initial monadic expression. Instead of generating code that returns a suspended computation, the compiler switches to generating code the invokes all subsequent monadic values.

The block `m205` in Figure 4.8 (b) (representing the body of the *kleisli* function) illustrates this principle. Though *kleisli* ends with a monadic expression ($f\ v$), it does not return a suspended computation representing $f\ v$; instead, it returns the result of executing *f v*. The statement on Line 9 evaluate `f @ v1` and assigns the result to `v206`. The next line invokes the thunk, immediately executing the program returned by `f @ v1`. This is a case where the compiler generates code to execute a sequence of monadic values, rather than code that returns a suspended computation.

4.7 Executing MIL Programs

As stated in Section 4.2, MIL's design borrows heavily from three-address code, an intermediate form that closely resembles an idealized assembly-language code. The execution model for MIL draws on its three-address code inspiration and executes like an assembly-language for a simple register-based computer: execution begins at a special designated point in the program and proceeds sequentially. When the first block executed in the program "returns," the program terminates.

Unlike assembly-language, MIL blocks also act like functions. Each block declares parameters and those names are only in scope over the block. Blocks always return a monadic value. Closure, thunk, and data allocations already create monadic values. The **`return`** keyword creates a computation that evaluates to the given value. The value produced by the last statement in a given block is returned to the block's caller.

Within each block, any number of storage locations may be named on the left-hand side of a bind (`<-`) statement. Those names are not global storage locations: variables with the same name in different blocks do not affect each other. Values can only be passed from one block to another as arguments, in a closure, or in a monadic thunk.

When an `@`, **`invoke`**, or goto expression appears on the right-hand side of a bind statement, control-flow transfers to the appropriate block. For `@` and `invoke`, the label stored in the closure or thunk determines the block to execute. For goto, the block is named directly. The value returned from the block is bound to the variable on the left-hand side of the bind statement.

Mil's syntax does not mention error handling or exceptions at all. However, errors can arise in a number of areas, such as when no case alternatives match the inspected value, or if a block is called with the wrong number of arguments. In all cases of errors, we expect that an error would be generated by the mil runtime or interpreter and that the program would terminate. More robust error handling would certainly be an area for improvement in the future.

4.8 A Hoopl-friendly AST For MIL

As described in Section 3.2, hoopl uses the *O* and *C* types to express the shape of the entry and exit points for a node. A node that is open on exit can only be followed by a node that is open on entry. A sequence of nodes can be characterized by the entry shape of the first node and the exit shape of the last node.

In hoopl terms, basic blocks (described in Section 2.2) are closed on entry and closed on exit. Due to the constraints imposed by the shape type, none of the nodes between the first and last will jump or branch to other nodes; they can only execute one after another.

Figure 4.9 shows *Stmt*, the data type defining the *block*, *bind*, and *done* terms from Figure 4.2 (Page 58). The *Stmt* type takes two type parameters, *e* and *x*, representing the entry and exit shape of the statement. *BlockEntry* and *CloEntry* represent the two types of blocks (basic and closure-capturing, respectively). Their shape, *C O*, shows that they can only be used to begin a mil block. The *Name* and

Label arguments help HOOPL connect nodes together in the CFG. The *Bind* statement (with shape *O O*) represents statements inside the block. The type ensures no block begins or ends with a *Bind*. Blocks can end with either a *Case* or *Done* statement. The *Case* value represents the `case` statement. The [*Alt Tail*] argument to *Case* lists each case alternative and provides a tail value to execute when the alternative is matched. The AST uses *Done* to end a block with a *Tail* expression. The *Name* and *Label* arguments to *Case* and *Done* make it easier to know the basic block being analyzed when traversing the CFG backwards, as HOOPL does not provide that information to backwards rewrite or transfer functions.[7]

```
data Stmt e x where
  BlockEntry :: Name → Label → [Name] → Stmt C O
  CloEntry :: Name → Label → [Name] → Name → Stmt C O
  Bind :: Name → Tail → Stmt O O
  Case :: Name → [Alt Tail] → Stmt O C
  Done :: Name → Label → Tail → Stmt O C
```

Figure 4.9: Haskell data type representing MIL *block*, *bind*, and *done* terms. The *C* and *O* types (from HOOPL) give the "shape" of each statement.

Figure 4.10 shows the *Tail* values that represent the *tail* terms in Figure 4.2. Notice the definition does not parameterize on shape. These values do not specify the relationship among blocks in the CFG, as HOOPL understands it, so we do not give them a shape type. *Return* and *Invoke* represent the corresponding *tail* terms. *Enter* corresponds to the `@` operator. *Goto* and *Prim* represent the labeled jumps

[7] We discuss this issue, and suggest improvements, in Section 6.3.2.

```
data Tail = Return Name
    | Enter Name Name
    | Goto Dest [Name]
    | Prim Name [Name]
    | Closure Dest [Name]
    | Thunk Dest [Name]
    | Invoke Name
    | Constr Constructor [Name]
```

Figure 4.10: Haskell data type representing *tail* terms.

$\mathtt{b}\,(\mathtt{v}_1,\ldots,\mathtt{v}_n)$ and $\mathtt{p}^*(\mathtt{v}_1,\ldots,\mathtt{v}_n)$. *Closure* allocates a closure ($\mathtt{k}\,\{\mathtt{v}_1,\ldots,\mathtt{v}_n\}$), *Thunk* allocates a monadic thunk ($\mathtt{b}\,[\mathtt{v}_1,\ldots,\mathtt{v}_n]$), and *Constr* allocates a data value. The *Constructor* type in *Constr* names the tag used for the data value. The *Dest* type in *Goto*, *Closure* and *Thunk* identifies the block associated with the term. Notice that *Prim* does not specify a *Dest* because there is no block (implemented in MIL) associated with a primitive function. Instead, *Prim* just specifies the primitive's name.

All *tail* terms allow only variables as arguments, not arbitrary expressions. The *Tail* constructors enforce that restriction by only taking *Name* arguments. Similarly, *Stmt* constructors only define arguments that are *Names* or *Tails*.

Recall **m205** from Figure 4.8 (b) on Page 68, which implements the body of our monadic composition function:

```
m205(g, x, f):
  v207 <- g @ x
  v1 <- invoke v207
  v206 <- f @ v1
  invoke v206
```

We can now show the *Stmt* and *Tail* values that represent `m205`:

```
m205 :: Label → Graph Stmt C C
m205 label =
  mkFirst (BlockEntry "m205" label ["g","x","f"]) <*>   -- m205 (g,x,f):
  mkMiddles [Bind "v207" (Enter "g" "x")                --   v207 <- g @ x
    ,Bind "v1" (Invoke "v207")                          --   v1 <- invoke v207
    ,Bind "v206" (Enter "f" "v1")] <*>                  --   v206 <- f @ v1
  mkLast (Done "m205" label (Invoke "v206"))            --   invoke v206
```

m205 defines a basic block, as shown by its *C C* type. Hoopl provides *mkFirst*, *mkMiddles*, and *mkLast* (Chapter 3, Figure 3.5) for lifting nodes into hoopl's monadic graph representation. The operator <*> connects pieces of the graph together. Hoopl uses the *label* argument to connect this definition to other basic blocks in a larger program.

The *BlockEntry* value defines the block. The *label* argument will be used elsewhere to refer to this block when manipulating or traversing the cfg built by hoopl. In fact, the *Dest* type in Figure 4.10 is a tuple containing the label of a given block and its name. The list of variables (["g","x","f"]) defines the arguments in scope at the beginning of the block. The three *Bind* statements that make up the body of the block follow *BlockEntry*. The comment to the left of each statement shows the operation implemented by that statement. Finally, the *Done* statement at the end of the block shows that the result of the block will be the value returned when the monadic thunk `v206` executes.

4.9 MIL CFGs with Hoopl

As discussed in Section 3.2, hoopl uses the *NonLocal* typeclass to define the successor and predecessor relationships among nodes in a cfg. Figure 4.11 gives the *NonLocal* instance for MIL. The *entryLabel* definitions show that the *Label* argument to *BlockEntry* and *CloEntry* serve to index MIL blocks in the CFG. That is, HOOPL uses *Label* values to find a given block.

```
instance NonLocal Stmt where
  entryLabel (BlockEntry _ l _) = l
  entryLabel (CloEntry _ l _ _) = l
  successors (Case _ alts) = [l | (Alt _ _ (Goto (_,l) _)) ← alts]
  successors (Done _ _ (Goto (_,l) _)) = [l]
  successors _ = [ ]
```

Figure 4.11: MIL's instance definition for *NonLocal*.

The *successors* definition shows that we only consider the last statement in a block when specifying successors. Recall that each *Case* alternative will immediately jump to some block; when a block ends with a *Case*, we consider all those blocks successors. When a block ends with a *Goto*, we specify the destination block as the sole successor to the block. The block will have no successors for all other *Tail* values.

Crucially, we do *not* consider any MIL blocks mentioned in *Goto* expressions on the right-hand side of a *Bind* statement as successors. This choice can make certain transformations unsound if we are not careful, because some control-flow facts will not be apparent to HOOPL. However, we choose this representation in order to allow support for the monadic transformations described in Section 6.1 on Page 119. We describe the scenarios that can lead to unsound transformations in Section 5.8.1 on Page 112.

4.10 Related Work

Mil builds on the large body of research and experience focused on monads and their use in the Haskell programming language. Our work is not the first to use a monadic intermediate language; previously published by Benton, Kennedy and others describes a monadic intermediate language for Standard ML (sml). We briefly describe monadic programming in Section 4.10.1, and discuss its relationship to our work. In Section 4.10.2 we discuss Benton and Kennedy's work, and draw some comparisons with our own.

4.10.1 Monads & Haskell

Moggi (1991) proposed monads as a means to model computation in real programs with side-effecting behaviors. Wadler popularized this notion as a way to structure functional programs; in particular, as a way to allow side-effecting computation in the "purely" functional λ-calculus.

Wadler (1990) described a way to translate the call-by-value λ-calculus into computations in a particular monad. His notation uses a '$*$' to represent the translation of a λ-calculus term to some monadic term, represented by M. For example, he represents the translation of a λ expression as:

$$(\lambda x.\, v)^* = [(\lambda x.\, v^*)]^M .$$

Notice the $*$ moves inside the λ on the right, meaning the body of the λ will be recursively translated. This scheme provided a special inspiration for mil; it essentially gives the translation of λ terms to closure-capturing blocks:

$$\lambda x.\, v = \texttt{k1}\,\{\}\,\texttt{x:}\; v.$$

Mil differs, however, by treating the allocation of closures and other data structures as side-effecting computation. In Wadler's scheme, those operations remain pure.

4.10.2 MLj & SML.NET

Benton, Kennedy and colleagues (1998) implemented mlj, a compiler for Standard ML (sml) that targeted the Java Virtual Machine. Benton, Kennedy, and a different group of co-authors later implemented sml.net (2004, 2005), another sml compiler that targeted Microsoft's Common Language Runtime. Both compilers first translated sml into a typed mil. The authors did not publish a description of the mil for sml.net, but their 1998 paper gives extensive details for the mil used by mlj. They do not use the term, but we will call their intermediate language milj, to distinguish it from our mil.

Benton and colleagues designed a sophisticated type system for milj; our mil does not use types. Their type system represents several side-effects, including allocations. They used type-directed optimization to eliminate side-effecting dead-code that only allocates; we can do the same over mil blocks using dead-code analysis that determines if, in the binding `v <-` *tail*, `v` is dead and *tail* is a closure, data, or thunk allocation.

Milj represents monadic binding as $\mathbf{do}\ \{x \leftarrow m_1; m_2\}$, where m_1 and m_2 can be any milj expression. This representation allows nested monadic terms like $\mathbf{do}\ \{x \leftarrow \mathbf{do}\ \{y \leftarrow m_1; m_2\}; m_3\}$. Kennedy (2007) shows how these terms can be optimized by exploiting associativity. For example, the term above becomes $\mathbf{do}\ \{y \leftarrow m_1; x \leftarrow m_2; m_3\}$. However, he also shows that the transformation can be needed $O(n^2)$ times, where n is the number of terms (i.e., x) bound, depending on the order in which this transformation is interleaved with others.

Our MIL does not express nested monadic computations as in MILj. The MILj program $\mathbf{do}\ \{x \leftarrow \mathbf{do}\ \{y \leftarrow m_1; m_2\}; m_3\}$ becomes:

```
b(...):
  y <- m1(...)
  x <- m2(...)
  $m_3$
```

where `m1` and `m2` implement m_1 and m_2; the translation of m_3 depends on the form of m_3, of course, so we leave it unspecified. In our representation, m_1 and m_2 become separate MIL blocks.

The transformation that Kennedy describes appears as inlining in MIL; that is, for an appropriate `m1`, we can inline the block into `b`. It is an open question if inlining for non-recursive blocks in MIL can be achieved in better than $O(n^2)$ time.

4.11 Summary

This chapter presented our Monadic Intermediate Language (MIL). MIL resembles three-address code in several ways: arbitrarily many registers can be named, nested expressions are not allowed, and implementation details are made explicit. The MIL's unique features include separate representations for *closure-capturing* and basic blocks, and the use of monadic *tail* expressions. Though we have not included the translation here, our implementation of a compiler from λ_C to MIL gives us confidence that every λ_C program can be represented in MIL.

Chapter 5

Uncurrying

Many functional languages allow programmers to write definitions that take advantage of *partial application*. Partial application means to give a function only some of its arguments, resulting in a new function that takes the remaining arguments. A function definition that supports partial application is said to be in *curried* style. In contrast, an *uncurried* function is defined such that it can only be applied to all of its arguments at once.

Partial application can be very convenient for programmers, but it can also be very inefficient. Conceptually, an uncurried function can do real work with each application — that is, each application executes the body of the function. A curried function does not do any real work until given all its arguments; each in-between application essentially creates a new function.

This chapter describes our implementation of *uncurrying*, an optimization that reduces the number of partial applications in a program. Through dataflow analysis, we find partial applications for a given function within a block of MIL code. We replace those partial applications with full applications to the function, or at least fewer partial applications.

Section 5.1 describes partial application in more detail; Section 5.2 discusses drawbacks to supporting partial application. We introduce several examples that will be used to illustrate our optimization in Section 5.3 and discuss uncurrying as applied to MIL in Section 5.4. We present our dataflow equations for uncurrying in Section 5.5 and our rewriting strategy in Section 5.6. We show our implementation in Section 5.7. We give two extended examples in Section 5.8, demonstrating our optimization's utility on more complicated CFGs. Many other implementations of uncurrying have been described elsewhere; we discuss those in Section 5.9. Section 5.10 summarizes our contribution.

5.1 Partial Application

Partial application in functional programming promotes reusability and abstraction. It allows the programmer to define specialized functions by fixing some of the arguments to a general function.

```
map1 :: (a → b) → [a] → [b]
map1 f xs = ...
```

Figure 5.1: A Haskell definition in curried style. *map1* can be partially applied directly to produce specialized functions.

For example, the Haskell code in Figure 5.1 defines *map1* in curried style. We can create specialized mapping functions by applying *map1* to a single argument. The following functions convert all their arguments to uppercase or square all integers in a list, respectively:

```
upCase1 :: [Char] → [Char]
upCase1 = map1 toUpper
square1 :: [Int] → [Int]
square1 = map1 (ˆ2)
```

5.2 Cost of Partial Application

At the assembly language level, function application is expensive because multiple operations must take place to implement it: saving registers, loading addresses, and finally jumping to the target location. Partial application exaggerates all these costs by essentially creating a *series* of functions, each of which takes one argument and returns a closure that points to the next function in the chain. Only when all the arguments are gathered does the function do "real work" — that is, something besides allocating closures and gathering up arguments.

Partial application also influences the code generated to implement function application. Rather than generate specialized code for partially versus fully-applied functions, it is simplest to generate the same code for all applications, partial or otherwise; meaning every function application pays the price of partial application, even if the function is "obviously" fully-applied.

5.3 Partial Application in MIL

Recall that MIL defines two types of blocks: "closure-capturing" and "normal." Normal blocks act much like labeled locations in a program and are written "**b** $(\mathtt{v}_1, \ldots, \mathtt{v}_\mathtt{n})$: ..." A normal block is executed by writing "**b** $(\mathtt{v}_1, \ldots, \mathtt{v}_\mathtt{n})$."

Closure-capturing blocks are also like labeled locations, except that they expect to receive a closure and an argument when called. We write closure-capturing blocks as "**k** $\{\mathtt{v}_1, \ldots, \mathtt{v}_\mathtt{n}\}$ x: ..." A closure-capturing block is always executed as the result of an expression like "f @ x."

These two definitions allow MIL to represent function application uniformly. For a function with *n* arguments, *n* closure-capturing blocks and at least one basic block will be generated. The first $(n-1)$ closure-capturing blocks are typically of the form:

$$\texttt{k}_\texttt{i}\,\{\texttt{v}_1,\ \ldots,\ \texttt{v}_\texttt{i}\}\,\texttt{x:}\ \texttt{k}_{\texttt{i}+1}\,\{\texttt{v}_1,\ \ldots,\ \texttt{v}_\texttt{i},\ \texttt{x}\}$$

This means the block $\texttt{k}_\texttt{i}$ returns a new closure that points to the next block ($\texttt{k}_{\texttt{i}+1}$) and contains all the values from the original closure as well as the argument `x` ($\{\texttt{v}_1,\ldots,\texttt{v}_\texttt{i},\texttt{x}\}$).

The last block, $\texttt{k}_{\texttt{n}-1}$, does not immediately return a new closure, but instead calls a basic block, `b`, with all necessary arguments. In the general case, we write $\texttt{k}_{\texttt{n}-1}$ as:

$$\texttt{k}_{\texttt{n}-1}\,\{\texttt{v}_1,\ \ldots,\ \texttt{v}_{\texttt{n}-1}\}\,\texttt{x:}\ \texttt{b}\,(\texttt{v}_1,\ \ldots,\ \texttt{v}_{\texttt{n}-1},\ \texttt{x})$$

Of course, depending on the definition of the original function, we may not pass all arguments to `b`, or pass them in the same order as they appear in the closure.

For example, Figure 5.2 (a) shows the λ_C definition for *compose* and its implementation in MIL.[1] The basic block `k0` acts as the top-level entry point to *compose*. The other basic block, `compose`, implements the body of *compose*. The two closure-capturing blocks, `k1` and `k2` implement MIL's support for partial application. The remaining closure-capturing block only executes when all of the arguments to *compose* are available.

Executing `k1` results in a closure that captures the argument `f` and points to `k2`. The closure returned is equivalent to the expression *compose a*, with *a* being the value held by the closure. Executing `k2` returns a closure that captures

[1] This same program also appears in Figure 4.5 on Page 64.

$compose\,f\ g\ x = f\ (g\ x)$

(a)

```
k0(): k1{}
k1{}f: k2{f}
k2{f}g: k3{f, g}
k3{f, g}x: compose(f, g, x)

compose(f, g, x): ... as in Figure 4.3 (b) on Page 61...
```

(b)

Figure 5.2: The *compose* function. Part (a) shows our λ_C definition. Part (b) shows MIL code implementing Part (a).

two values, f and g, and points to **k3**. The closure returned is equivalent to the expression *compose a b*, with *a* and *b* held by the closure. The values returned by these two blocks represent partially applied functions. The remaining closure-capturing block, **k3**, does not return a value representing a partially applied function, however.[2] Instead, **k3** immediately executes the **compose** block, and is the same as evaluating *compose a b c*.

5.4 Uncurrying MIL blocks

Using the definition of *compose* given in Figure 5.2 (b), we can give a MIL implementation of a partially-applied *compose* function, *compose1 f* = *compose f*:

```
compose1(f):
  t0 <- k0()
  t0 @ f
```

[2] Unless, of course, *compose a b c* results in a function value!

Examination of this definition reveals one opportunity for optimization: the call "k0 ()" on the first line assigns the closure k1 {} to t0, which we immediately enter on the next line with argument f. We can eliminate the call to k0 by allocating the closure directly:

```
compose1 (f):
  t0 <- k1 {}
  t0 @ f
```

Now we can see that t0 holds the value k1 {}, a closure referring to block k1 and capturing no variables. Block k1 also returns a closure, this time capturing its argument and pointing to block k2. With this knowledge, we can eliminate the expression t0 @ f and instead create the closure directly, using the expression k2 {f}:

```
compose1 (f):
  t0 <- k1 {}
  k2 {f}
```

Now we find that t0 is no longer used, allowing us to rewrite compose1 one more time:

```
compose1 (f): k2 {f}
```

Thus, by uncurrying, we eliminate one call (k0 ()), one enter operation (t0 @ f), and the creation of one closure (k1 {}).

Our uncurrying optimization transforms MIL programs to eliminate @ operations as we did by hand for *compose1*. In essence, we determine if an @ operation results in a known closure, allowing us to replace that expression with the closure returned.

5.5 Dataflow Equations

We implement uncurrying with a forwards dataflow analysis. Our facts indicate if a given variable refers to a known closure. Facts are propagated to successor blocks when the block ends with a call or case statement. We combine multiple input facts for a given block by determining if all sets of facts agree on the value of a given variable.

Figure 5.3 shows the dataflow equations used for our analysis. The sets Labels and Vars contain all labels and all variables in the program, respectively. The Clo set associates some label with a (possibly empty) list of variables. We use Clo values to represent the location that a closure points to and the set of variables that it captures. The Fact set defines the facts that we can compute, each of which is a pair, $(\mathtt{v}, p)$, associating a bound variable v with a value p. If $p \in \textsc{Clo}$, then v refers to a known location and an associated set of captured variables. Otherwise, if $p = \top$, then v refers to some unknown value.

We combine sets of Fact values using a meet operator, $\wedge$, as defined in Equation (5.2), over two sets of facts, F_1 and F_2. When a variable v only appears in F_1 or F_2, we assume we do not not know what value v may represent, so we add $(\mathtt{v}, \top)$ to the result. When a variable appears in both F_1 and F_2, we create a new pair by combining the two associated Clo values using the $\sqcap$ operator defined in Equation (5.1). The resulting pair has the same variable but a (possibly) new Clo value. Together, Fact and $\wedge$ form a lattice as described in Section 2.3.1 on Page 9.

For example, if $F_1 = \{(\mathtt{v}, \mathtt{l}\,\{\mathtt{a}\}), (\mathtt{w}, \mathtt{l}\,\{\mathtt{b}\})\}$ and $F_2 = \{(\mathtt{u}, \mathtt{l}\,\{\mathtt{a}\}),\ (\mathtt{v}, \mathtt{l}\,\{\mathtt{a}\}),\ (\mathtt{w}, \mathtt{l}\,\{\mathtt{a}\})\}$ then $F_1 \wedge F_2$ would be $\{(\mathtt{u}, \top), (\mathtt{v}, \mathtt{l}\,\{\mathtt{a}\}), (\mathtt{w}, \top)\}$. Because u only appears in one set, we cannot assume it will always refer to $\mathtt{l}\,\{\mathtt{a}\}$, so we add the pair $(\mathtt{u}, \top)$

Facts

$$\begin{aligned}
\textsc{Labels} &= \text{Set of all program labels.}\\
\textsc{Vars} &= \text{Set of all variables.}\\
\textsc{Clo} &= \{\mathtt{b}\,\{\mathtt{v_1},\ldots,\mathtt{v_n}\} \mid \mathtt{b} \in \textsc{Labels}, \mathtt{v_i} \in \textsc{Var}, n \geqslant 0\}\\
\textsc{Fact} &= \textsc{Vars} \times (\{\top\} \cup \textsc{Clo}).
\end{aligned}$$

Meet

$$p \sqcap q = \begin{cases} p & \text{when } p = q\\ \top & \text{when } p \neq q, \end{cases} \tag{5.1}$$

where $p, q \in \textsc{Clo}$.

$$\begin{aligned}
F_1 \wedge F_2 \;=\; &\{(\mathtt{v}, p \sqcap q) \mid (\mathtt{v}, p) \in F_1, (\mathtt{v}, q) \in F_2\} \cup\\
&\{(\mathtt{v}, \top) \mid \mathtt{v} \in dom(F_1), \mathtt{v} \notin dom(F_2) \vee\\
&\qquad\quad \mathtt{v} \notin dom(F_1), \mathtt{v} \in dom(F_2)\},
\end{aligned} \tag{5.2}$$

where $F_1, F_2 \in \textsc{Fact}$.

Transfer Function

$$t(F, \mathtt{b}\,(\ldots)\mathtt{:}) = F; \quad t(F, \mathtt{k}\,\{\ldots\}\,\mathtt{x:}) = F \tag*{(5.3), (5.4)}$$

$$t(F, \mathtt{v} \;\texttt{<-}\; \mathtt{b}\,\{\mathtt{v_1}\}) = \begin{cases} (F \cup \{(\mathtt{v}, \mathtt{b}\,\{\mathtt{v_1}\})\}) \setminus uses(F, \mathtt{v}) & \text{when } \mathtt{v} \neq \mathtt{v_1}\\ delete(F, \mathtt{v}) & \text{when } \mathtt{v} = \mathtt{v_1} \end{cases} \tag{5.5}$$

$$t(F, \mathtt{v} \;\texttt{<-}\; \ldots) = \{(\mathtt{v}, \top)\} \cup (F \setminus uses(F, \mathtt{v})) \tag{5.6}$$

$$t(F, \mathtt{b}\,(\mathtt{v})) = \{\mathtt{b} \;:\; rename(args(\mathtt{b}), \mathtt{v}, restrict(F, \mathtt{v}))\} \tag{5.7}$$

$$t\left(F, \begin{array}{l} \quad\texttt{case v of}\\ \quad\; \mathtt{C_1\ v_1 \;\texttt{->}\; b_1\,(w_1)}\\ \quad\vdots \end{array}\right) = \left\{\begin{array}{l} \mathtt{b_1} : rename(args(\mathtt{b_1}), \mathtt{w_1},\\ \qquad\qquad trim(restrict(F, \mathtt{w_1}), \mathtt{v_1}))\\ \qquad\qquad\quad \vdots \end{array}\right\} \tag{5.8}$$

$$t(F, \_) = \emptyset, \tag{5.9}$$

where $F \in \textsc{Fact}$.

$$uses(F, \mathtt{v}) = \begin{array}{l} \{(\mathtt{u}, \mathtt{l}\,\{\mathtt{v_1}, \ldots, \mathtt{v_n}\}) \mid\\ \quad (\mathtt{u}, \mathtt{l}\,\{\mathtt{v_1}, \ldots, \mathtt{v_n}\}) \in F, \mathtt{v} \in \{\mathtt{v_1}, \ldots, \mathtt{v_n}\}\} \end{array} \tag{5.10}$$

$$rename(\mathtt{u}, \mathtt{v}, F) = \begin{array}{l} \{(\mathtt{u}, p) \mid (\mathtt{v}', p) \in F, \mathtt{v} = \mathtt{v}'\}\\ \qquad\qquad \cup \{(\mathtt{v}', p) \mid (\mathtt{v}', p) \in F, \mathtt{v} \neq \mathtt{v}'\} \end{array} \tag{5.11}$$

$$delete(F, \mathtt{v}) = \{(\mathtt{u}, p) \mid (\mathtt{u}, p) \in F, \mathtt{u} \neq \mathtt{v}\} \tag{5.12}$$

$$restrict(F, \mathtt{v}) = \{(\mathtt{v}', p) \mid (\mathtt{v}', p) \in F, \mathtt{v} = \mathtt{v}'\} \tag{5.13}$$

$$trim(F, \mathtt{v}) = delete(F \setminus uses(F, \mathtt{v}), \mathtt{v}) \tag{5.14}$$

$$args(\mathtt{b}) = \textit{\ldots set of parameters declared by } \mathtt{b}\ldots, \tag{5.15}$$

where $F \in \textsc{Fact}, \mathtt{v} \in \textsc{Var}$.

Figure 5.3: Dataflow facts and equations for our uncurrying transformation.

to the result. The variable `v` appears in both sets with the same closure, so we add $(\mathtt{v}, \mathtt{l}\,\{\mathtt{a}\} \sqcap \mathtt{l}\,\{\mathtt{a}\})$, or $(\mathtt{v}, \mathtt{l}\,\{\mathtt{a}\})$, to the result set. Finally, `w` appears in both sets, but the closure associated with it in each differs: $\mathtt{l}\,\{\mathtt{b}\}$ in F_1 and $\mathtt{l}\,\{\mathtt{a}\}$ in F_2. Therefore, we add $(\mathtt{w}, \top)$ to the result set.

Our transfer function, t, takes a statement and a set of Fact values as arguments. It returns a Fact set containing new facts based on the statement given. We define t by cases over MIL statements.

Equations (5.3) ***and*** (5.4) — ***Block Entry*** These equations represent the entry points for normal and closure-capturing blocks. We do not modify the facts received, but just pass them along to the next statement in the block.

Equation (5.5) — ***Bind To Closure*** When the right-hand side of a `<-` statement creates a closure, as in $\mathtt{v} \leftarrow \mathtt{b}\,\{\mathtt{v_1}, \ldots, \mathtt{v_n}\}$, we may or may not create a new fact. If `v` appears in $\{\mathtt{v_1}, \ldots, \mathtt{v_n}\}$ (as in "$\mathtt{v} \leftarrow \mathtt{k1}\,\{\};\ \mathtt{v} \leftarrow \mathtt{k2}\,\{\mathtt{v}\}$"), then we simply delete any facts mentioning `v` and we do not create a new fact. Otherwise we create the fact $(\mathtt{v}, \mathtt{b}\,\{\mathtt{v_1}, \ldots, \mathtt{v_n}\})$. Because `v` has been redefined, we must invalidate any previous facts that refer to `v`, as they do not refer to the new value of `v`. To ensure we remove all references to `v`, we apply *uses* to the combined set $F \cup \{(\mathtt{v}, \mathtt{b}\,\{\mathtt{v_1}, \ldots, \mathtt{v_n}\})\}$. We subtract the result from F, thereby removing any facts that refer to `v`.

Equation (5.6) — ***Any Other Bind*** Any other binding, $\mathtt{v} \leftarrow \ldots$, that does not create a closure invalidates any facts about `v`. Therefore, we first remove all facts referring to `v` in F with the *uses* function. We then create a new fact associating `v` with $\top$, indicating that we know `v` does not refer to a closure.

Finally, we combine the new set and new fact and return the combined set.

Mil blocks that end with a `case` statement can have multiple successors. Dataflow analysis does not usually specify that different facts go to different successors, but we do so here. The notation $\{\mathtt{b}_1 : F_1, \mathtt{b}_2 : F_2\}$ used in Equations (5.7) and (5.8) means we transfer the set of facts F_1 to the successor block $\mathtt{b}_1$, and the set of facts F_2 to the successor block $\mathtt{b}_2$.

Mil blocks also specify formal parameters, and the names of those parameters usually differ from the actual variables used in a given "goto" expressions. Within a block, we collect facts using the names local to that block. Those facts will have no meaning in successor blocks (or worse, the wrong meaning) because the variable names will differ. Equation 5.11 defines *rename*, which takes a set of facts, F, and two variables, $\mathtt{u}$ and $\mathtt{v}$. If a fact about $\mathtt{v}$ exists in F, we update it to be about $\mathtt{u}$. Combined with the *args* function, which retrieves the list of formal parameters for a block, *rename* can update a set of facts from one block so that it makes sense in a successor block.

The two equations,(5.7) and (5.8), describe how we transfer facts between blocks using the functions given above. In this presentation, we only show one variable, but the equations can be easily extended to a multiple variables. We also use a number of auxiliary definitions, besides those mentioned above. The *trim* function applies the *uses* and *delete* functions to remove all facts from F that refer to or are about $\mathtt{v}$. The *delete* function removes any facts about $\mathtt{v}$ from F. Conversely, the *restrict* function filters all facts from F except those about $\mathtt{v}$.

Equation (5.7) — ***Goto Block*** When a "goto" expression, such as $\mathtt{b}\,(\mathtt{v})$, appears at the end of a block, we transfer the facts collected so far to the successor block.

We use the *restrict* function to remove all facts from F except those about `v`. We then rename the facts to match the successor block `b`, and pass those facts along to `b`.

Equation (5.8) — ***Case Statement*** A case statement requires careful treatment. Recall that each alternative arm jumps immediately to another block ($\texttt{b}_1$, etc. in the equation). We pass separate sets of facts to each successor, tailored to the arguments that each block declares. Additionally, the alternative can bind new variables, shadowing previous bindings. Any of our existing facts that are about or that refer to shadowed variables must be removed from our facts before we pass them to successor blocks.

For each successor block $\texttt{b}_\texttt{i}$, we first restrict our facts to include only those variables passed to the block (i.e., $\texttt{w}_\texttt{i}$). From that restricted set, we trim any facts that mention a binding from the case alternative (i.e., $\texttt{v}_\texttt{i}$). Finally, we rename those facts according the formal arguments of the successor block $\texttt{b}_\texttt{i}$. We stress that, while these equations only mention one variable in the alternative and in the call to $\texttt{b}_\texttt{i}$, making an operation like *trim* trivial, they can easily be extended to multiple variables, allowing them to be used with real MIL programs.

Equation (5.9) — ***All Other Statements*** Our final equation covers all other types of expression that can appear at the end of a block, such as a function application or allocation. None of these expressions specify a successor block, so in a sense it does not matter what they return as that value will be ignored. For completeness, however, we return the empty set in this final case.

5.6 Rewriting

The facts gathered by our dataflow analysis allow us to replace `@` expressions with closure allocations if we know the value that the expression results in. For example, let F be the facts computed so far and `v <- f @ y` the statement we are considering. If $(\texttt{f}, \texttt{k0}\,\{\texttt{x}\}) \in F$, then we know `f` represents the closure `k0` {x}, and we may be able to rewrite the expression. If `k0` returns `k1` {x, y}, then we can rewrite the statement to `v <- k1` {x, y}. Alternatively, if `k0` immediately calls `b0` (x, y), we can rewrite the statement to `v <- b0` (x, y). In both cases it is likely that the formal arguments to `k2` differ from those in either the closure `k0` {x} or the expression `f @ y`, and we will need to rename our facts. However, as explained previously when discussing t, that is a straightforward operation.

The example we discussed in Section 5.4 does not match the optimization just discussed on one crucial point: replacing calls to normal blocks on the right-hand side of a `<-` with their closure result. Our implementation relies on another, more general, optimization that inlines simple blocks into their predecessor. We discuss the optimization in Chapter 6, Section 6.1.1 on Page 120, but in short that optimization will inline calls to blocks such as `compose`, so a statement like `v <- compose` () becomes `v <- absBodyL201` {}, where `absBodyL201` is the label in the closure returned by `compose` ().

5.7 Implementation

Originally, we called this transformation "closure-collapse" because it "collapses" the construction of multiple closures into the construction of a single closure. Later, we learned that this optimization is known as "uncurrying," but at the point the code had already been written. The “collapse” prefix in the code shown is an artifact of our previous name for the analysis.

Figure 5.4 gives an example program that we will use throughout this section to illustrate our implementation. The program takes a string as input, converts it to an integer, doubles that value, and returns the result. The program consists of five blocks. Two of the blocks, `k0` and `k1`, are closure-capturing. Two others, `add` and `toInt`, are normal blocks that call runtime primitives. The final block, `main`, is also a normal block but is treated as the entry point for the program.

```
main(s):
  n <- toInt(s)
  v0 <- k0{}
  v1 <- v0  @  n
  v2 <- v1  @  n
  return v2

k0{} a: k1{a}
k1{a} b: add(a, b)
add(x, y): plus*(x, y)
toInt(s): atoi*(s)
```

Figure 5.4: A mil program we will use to illustrate our implementation of uncurrying.

We present our implementation in five sections, reflecting the structure of our dataflow equations above. We first give the types used, followed by the definition of our lattice, then our transfer function, then our rewriting function, and finally we show the driver that applies the optimization to a given program.

5.7.1 Types

Figure 5.5 shows the types used by our implementation to represent the sets given in Figure 5.3. The *Clo* type represents Clo. *Label* and *Var* correspond to Label and Var, respectively. For documentation, the *Label* type pairs a string with hoopl's *Label* type. *Clo* stores a *Label* value, giving the block that the closure refers to, and a list of captured variables, [*Var*], representing the environment captured by the closure.

```
data Clo = Clo Label [Var]
type Label = (String, Hoopl.Label)
type Var = String
data DestOf = Jump Label [Int] | Capture Label Bool
type Fact = Map Var (WithTop Clo)
```

Figure 5.5: The types for our analysis. Referring to the sets defined in Figure 5.3, *Clo* represents Clo and *Fact* represents Fact. *DestOf* is not represented in our dataflow equations; it describes the behavior of each mil block that we may use while rewriting.

The *DestOf* type captures the behavior of a given closure-capturing block. Recall that we limit closure-capturing blocks to containing a single *tail* expression. The *DestOf* type uses the *Capture* and *Jump* constructors to indicate if the block returns a closure or if it jumps to a normal block, respectively. The *Label* value in both is a destination: either the label stored in the closure returned, or the block that the closure jumps to. We use these values to determine how we rewrite a given "enter" expression.

The *Capture* value represents a block with the form "k0 $\{v_1, \ldots, v_n\}$ x: k1 $\{\ldots\}$," The flag in the *Capture* constructor indicates if `k1` {...} includes the `x` argument or not. If *True*, the argument is included in the closure returned. Otherwise, the argument is ignored.

The *Jump* value represents a block with the form "`k` {$v_1, \ldots, v_n$} `x: b` (...)" The arguments to `b` are not necessarily in the same order as the parameters for the closure-capturing block `k`. Each integer in the list given to *Jump* corresponds to one of `k`'s parameters. The value of the integer gives the position of that parameter in the call to `b`. The arguments in the call to `b` are built by traversing the list, putting the variable indicated by each index into the corresponding argument for the block.[3]

For example, in the following, the variables in the closure received by `c` do not appear in the same order as expected by block `l`:

```
c {a, b} x: l (x, a, b)
l (x, a, b): ...
```

We represent `c` using *Jump* `l` $[2, 0, 1]$, because the variables from the closure $\{a, b\}$ and the argument `x` must be given to `l` in the order (x, a, b).

Fact is a finite map, representing our FACT set. HOOPL's *WithTop* type adds a $\top$ value to any other type. *WithTop Clo* then represents the set $\{\top\} \cup \{\text{CLO}\}$. *Fact*, then, associates variables with values in the set $\{\top\} \cup \{\text{CLO}\}$.

[3] This situation can also apply to *Capture* blocks and we would need to update our implementation our compiler's code generation strategy changed or if we began writing MIL programs directly.

5.7.2 Lattice & Meet Operator

Figure 5.6 shows the *DataflowLattice* structure defined for our analysis. We set *fact_bot* to an empty map, meaning that we start without any information. We define *lub* over *Clos*, just like ⊓ in Figure 5.3. We use *joinMaps*, provided by HOOPL, and *toJoin* to transform *lub* into a function that operates over finite maps and that has the signature required by HOOPL's *fact_join* definition.

```
collapseLattice :: DataflowLattice Fact
collapseLattice = DataflowLattice {fact_name = "Closure collapse"
  ,fact_bot = Map.empty
  ,fact_join = joinMaps (toJoin lub)}
toJoin :: (a → a → (ChangeFlag, a))
    → (Hoopl.Label → OldFact a → NewFact a → (ChangeFlag, a))
toJoin f = \_ (OldFact o) (NewFact n) → f o n

lub :: WithTop Clo → WithTop Clo → (ChangeFlag, WithTop Clo)
lub (PElem (Clo l _)) new@(PElem (Clo l′ _))
    | l == l′ = (NoChange, new)
    | otherwise = (SomeChange, Top)
lub Top _ = (NoChange, Top)
lub _ _ = (SomeChange, Top)
```

Figure 5.6: The HOOPL *DataflowLattice* declaration representing the lattice used by our analysis.

5.7.3 Transfer Function

The definition of *transfer* in Figure 5.7 gives the implementation of *t* from Figure 5.3. The top-level definition, *collapseTransfer*, packages *transfer* into the *FwdTransfer* value that HOOPL uses to represent forwards transfer functions. The *blockParams* argument to *collapseTransfer* gives the list of parameters for every ordinary block in the program, which we use during renaming operations. The first argument to *transfer* is the statement we are analyzing, and the second is our facts so far. *transfer* depends on a number of auxiliary functions: *kill*, *using*, etc. We will describe each function as it is first encountered when describing *transfer*. The *Map* prefix on some of the functions used by *transfer* and related definitions indicates they are imported from Haskell's standard *Data.Map* library. We define *transfer* by cases, analogous to the cases given in Equations (5.3) through (5.9).

BlockEntry, *CloEntry* — These cases apply to the entry point of each normal or closure-capturing block, implementing Equations (5.3) and (5.4). In both instances they just pass the facts received on to the rest of the block.

Bind v (*Closure dest args*) — This case corresponds to Equation 5.5, representing a bind statement that allocates a closure on its right-hand side. Binding a variable invalidates any facts previously collected about that variable. The local definition of *facts*′ on Line 10 uses the *kill* function to remove all facts from *fact* that mention *v*. If *v* appears in *args* then the closure mentions the variable being bound. If that is the case, then we do not want to create a new fact, and we want to remove any existing facts about *v*. Line 8 accomplishes both tasks by first deleting any facts about *v* from *facts*′ and then returning the updated map. Otherwise, on Line 9 we create a new fact describing the closure (using hoopl's *PElem* constructor), insert it into *facts*′, and return the result.

Bind v _ — This case implements Equation 5.6. It removes any facts mentioning

```
collapseTransfer :: Map Hoopl.Label [Name] → FwdTransfer Stmt Fact
collapseTransfer blockParams = mkFTransfer transfer
  where
    transfer :: Stmt e x → Fact → Hoopl.Fact x Fact
    transfer (BlockEntry _ _ _) facts = facts
    transfer (CloEntry _ _ _ _) facts = facts
    transfer (Bind v (Closure dest args)) facts
        | v ‘elem‘ args = Map.delete v facts′
        | otherwise = Map.insert v (PElem (Clo dest args)) facts′
      where facts′ = kill v facts
    transfer (Bind v _) facts = Map.insert v Top (kill v facts)
    transfer (Done _ _ (Goto (_, dest) args)) facts = mapSingleton dest facts′
      where facts′ = rename args (blockParams ! dest) (restrict facts args)
    transfer (Case _ alts) facts = mkFactBase collapseLattice facts′
      where facts′ = [(dest, rename args params trimmed) |
                        (Alt _ binds (Goto (_, dest) args)) ← alts,
                        let trimmed = trim (restrict facts args) binds
                            params = blockParams ! dest]
    transfer (Done _ _ _) facts = mkFactBase collapseLattice [ ]
    kill :: Name → Fact → Fact
    kill = Map.filter . keep
    keep :: Name → WithTop Clo → Bool
    keep _ Top = True
    keep v (PElem (Clo _ vs)) = not (v ‘elem‘ vs)
    restrict :: Fact → [Var] → Fact
    restrict fact vs = Map.filterWithKey (\v _ → v ‘elem‘ vs) fact
    trim :: Fact → [Var] → Fact
    trim fact vs = foldr Map.delete (foldr kill fact vs) vs
    rename :: [Name] → [Name] → Fact → Fact
    rename args params = Map.mapKeys renameKey
      where renameKey v = maybe v (params!!) (v ‘elemIndex‘ args)
```

Figure 5.7: Our implementation of the transfer function *t* from Figure 5.3.

v and inserts a new fact associating *v* with *Top*, indicating we do not know what value *v* may have.

Done _ _ (*Goto* (_, *dest*) *args*) **—** On Line 12, we implement Equation 5.7. Recall that we must filter our facts to those about variables in *args*, and that we must rename those facts to match the parameters declared by the block represented by *dest*. The definition of *facts*′ (Line 13) uses the *restrict* function for filtering, and the *rename* function for renaming. We use HOOPL's *mapSingleton* function to create a set of facts associated with the block given by *dest*, analogous to the $\{\mathtt{b} : \ldots\}$ notation used in Equation 5.7.

Case _ *alts* **—** Recall that Equation 5.8 produced a map associating each successor block with a set of facts. The list comprehension on Lines 15–18 defines *facts*′ as a list of (*Label*, *Fact*) pairs. Each pair represents the facts passed to a given successor block. On Line 14, we apply HOOPL's *mkFactBase* function to *facts*′, returning a map associating each *Label* with a *Fact* set — just as in Equation 5.8.

Line 16 extracts each alternative from *alts*, the list of alternatives associated with the `case` statement. We defined MIL such that each alternative immediately jumps to a block; *dest* represents the destination block for the alternative, and *args* the variables passed to that block. Each alternative can introduce new bindings, represented here by the *binds* list. On Line 17, we use *restrict* to filter our set of facts to those about *args*. Because new bindings introduced by the alternative can invalidate existing facts, we use the *trim* function to remove any facts from the restricted set that are about or mention a variable in *binds*. Finally, we need to rename our facts to match the parameter names used by the successor block. On Line 18, we retrieve the parameter list for the given block. Line 15 uses the *rename* function to rename all facts in *trimmed* that are about variables in *args* to match the names given in *params*.

Done _ _ _ — A block that does not end in one of the cases above has no successors. Therefore, we just return an empty set of facts (as in Equation 5.9.). We construct an empty set by passing *mkFactBase* an empty list.

Statement	`n`	`v0`	`v1`	`v2`
`n <- ` **`toInt`** `(s)`				
`v0 <- ` **`k0`** `{}`	⊤			
`v1 <- v0 @ n`	·	**`k0`** `{}`		
`v2 <- v1 @ n`	·	·	⊤	
`return` `v2`	·	·	·	⊤

Figure 5.8: Facts about each variable in the **`main`** block of our example program from Figure 5.4. A blank entry means the variable has no facts associated with it yet. A "·" entry means the fact remains unchanged.

Figure 5.8 shows the facts gathered for each variable in the **`main`** block of our sample program, after the corresponding statement is analyzed. The variables `n`, `v1`, and `v2` are assigned ⊤ because the right-hand side of the `<-` statement for each does not directly create a closure. Only `v0` is assigned a *Clo* value, **`k0`** `{}`, because the right-hand side of its `<-` statement is in the correct form. We will see in the next section how these facts evolve as the program is rewritten.

5.7.4 Rewrite

Figure 5.9 shows the top-level implementation of our rewrite function for the uncurrying optimization. *collapseRewrite* creates the rewriter that can uncurry a MIL program. The *blocks* argument associates every closure-capturing block in our program with a *DestOf* value. *DestOf*, as explained in Section 5.7.1, indicates if the block returns a closure or jumps immediately to another block. The *rewrite* function actually implements the uncurrying transformation; we will describe it after discussing how we use HOOPL's iterative rewriting function, *iterFwdRw*.

```
1 collapseRewrite :: FuelMonad m ⇒ Map Hoopl.Label DestOf
2     → FwdRewrite m Stmt Fact
3 collapseRewrite blocks = iterFwdRw (mkFRewrite rewriter)
```

Figure 5.9: The top-level implementation of our uncurrying rewriter..

On Line 3, *collapseRewrite* applies HOOPL's *iterFwdRw* and *mkFRewrite* functions to create a *FwdRewrite* value. *iterFwdRw* applies *rewriter* repeatedly, until the *Graph* representing the program stops changing. Hoopl computes new facts (using *collapseTransfer*) after each rewrite. This ensures that a chain of closure allocations will be collapsed into a single allocation, if possible.

Figure 5.10 demonstrates this iterative process by showing how the `main` block in our example program changes over three iterations. The second column of each row shows facts computed for the program text in the first column. The value of *blocks* stays constant throughout, so we only show it once.

During the first iteration, *rewriter* transforms `v1 <- v0 @ n` to `v1 <- k1 {n}`, because `v0` holds the closure `k0 {}`, and *blocks* tells us that `k0` returns a closure pointing to `k1`.

Iteration	`main`	Facts	*blocks*
1	`n <- toInt (s)`	(n, ⊤),	`k0`: *Capture* `k1` *True*
	`v0 <- k0 {}`	(v0, `k0 {}`),	`k1`: *Jump* `add` [0, 1]
	`v1 <- v0 @ n`	(v1, ⊤),	
	`v2 <- v1 @ n`	(v2, ⊤)	
	`return` `v2`		
2	`n <- toInt (s)`	(n, ⊤),	
	`v0 <- k0 {}`	(v0, `k0 {}`),	
→	`v1  <- k1 {n}`	(v1, `k1 {n}`),	
	`v2  <- v1 @ n`	(v2, ⊤)	
	`return` `v2`		
3	`n <- toInt (s)`	(n, ⊤),	
	`v0 <- k0 {}`	(v0, `k0 {}`),	
	`v1 <- k1 {n}`	(v1, `k1 {n}`),	
→	`v2 <- add (n, n)`	(v2, `add (n, n)`)	
	`return` `v2`		

Figure 5.10: How *rewriter* transforms the `main` block. Each row represents `main` after the particular iteration. The first line shows the original program. The arrows shows the line that changed during each iteration. After the second iteration, the program stops changing.

The facts shown for the second iteration reflect the rewrite made, associating `v1` with `k1 {n}`. *rewriter* transforms `v2 <- v1 @ n` to `v2 <- add (n, n)` after this iteration because `v1` refers to `k1` and *blocks* tells us that `k1` jumps immediately to `add`. No changes occur after the third iteration because no statements remain that can be rewritten, and HOOPL stops applying *rewriter*. Note, however, that we could apply dead-code elimination at this point to remove `v0` and `v1`, because they are no longer referenced.

Figure 5.11 shows the functions that implement our uncurrying optimization.[4] Line 3 of *rewriter* rewrites `f @ x` expressions when they occur at the end of a block. Line 4 rewrites when `f @ x` appears on the right-hand side of a `<-` statement. In the first case, *done n l* (*collapse facts f x*) produces **`return`** *e* when *collapse* returns *Just e* (i.e., a rewritten expression). In the second case, *bind v* (*collapse facts f x*) behaves similarly, producing `v <-` *e* when *collapse* returns *Just e*. Both *done* and *bind* are defined in a separate file, not shown; they make it easier to construct *Done* and *Bind* values based on the *Maybe Tail* value returned by *collapse*. In all other cases, no rewriting occurs.

The *collapse* function takes a set of facts and two names, representing the left and right-hand arguments of the expression `f @ x`. When `f` is associated with a closure value, **`k`** {...}, in the *facts* map (Line 9), *collapse* uses the *blocks* argument to look up the behavior of the destination **`k`**. Lines 11 and 13 test if **`k`** returns a closure or jumps immediately to another block. In the first case, *collapse* returns a new closure-creating expression (*dest* {...}). In the second case, *collapse* returns a new goto expression (*dest* (...)).

If the destination immediately jumps to another block (Line 11), then we will rewrite `f @ x` to call the block directly. The list of integers associated with *Jump* specifies the order in which arguments were taken from the closure and passed to the block. *collapse* uses the *fromUses* function to re-order arguments appropriately.

In Figure 5.10, we showed that the *DestOf* value associated with **`k1`** is *Jump* **`add`** [0,1]. The list [0,1] indicates that **`add`** takes arguments in the same

[4] Note that these definition are local to *collapseRewrite*, so the *blocks* argument remains in scope.

```
1  rewriter :: FuelMonad m ⇒ forall e x . Stmt e x → Fact
2      → m (Maybe (ProgM e x))
3  rewriter (Done n l (Enter f x)) facts = done n l (collapse facts f x)
4  rewriter (Bind v (Enter f x)) facts = bind v (collapse facts f x)
5  rewriter _ _ = return Nothing
6  collapse :: Fact → Name → Name → Maybe Tail
7  collapse facts f x =
8    case Map.lookup f facts of
9      Just (PElem (Clo dest@(_, l) vs)) →
10       case Map.lookup l blocks of
11         Just (Jump dest uses) →
12           Just (Goto dest (fromUses uses (vs ++ [x])))
13         Just (Capture dest usesArg) →
14           Just (Closure dest
15             (if usesArg then vs ++ [x] else vs))
16         _ → Nothing
17     _ → Nothing
18 fromUses :: [Int] → [Name] → [Name]
19 fromUses idxs args = map (args!!) idxs
```

Figure 5.11: The implementation of our uncurrying rewriter.

order as they appear in the closure. However, if `add` took arguments in the opposite order, `k1` and `add` would look like the following code:

```
k1 {a} b: add(b, a)
add(x, y): ...
```

and the *DestOf* value associated with `k1` would be *Jump* `add` $[1, 0]$.

If the destination returns a closure (Line 13), then we rewrite `f @ x` to allocate the closure directly. The Boolean value *usesArg* indicates if the closure returned should capture the argument *x* or not.

5.7.5 Optimization Pass

Figure 5.12 presents *collapse*, which applies the uncurrying dataflow analysis and rewrite to the MIL program represented by the argument *program*. Line 3 analyzes and transforms *program* by passing appropriate arguments to HOOPL's *analyzeAndRewriteFwd* function. On Line 2, we evaluate HOOPL's monadic program using *runSimple*, which provides a monad with infinite optimization fuel.

Half of Figure 5.12 creates arguments for *analyzeAndRewriteFwd*, which we will detail in order.

fwd — This argument packages the lattice, transfer and rewrite definitions we described in Sections 5.7.2, 5.7.3, and 5.7.4.

JustC labels — We must give HOOPL all entry points for the program analyzed. These labels tell HOOPL where to start traversing the program graph. MIL does not define any particular block as an entry point, so all blocks in *program* will be analyzed. This argument's type is *MaybeC C* [*Label*], which requires us to use the *JustC* constructor.

program — This argument gives the program that will be analyzed and (possibly) transformed.

initial — The final argument gives initial facts for each label. Our analysis does not specify any prior knowledge at each label, so we set all initial facts to *Map.empty*. That is the value we gave *fact_bot* when defining our *Data flow - Lattice* value (Figure 5.6).

```
collapse :: ProgM C C → ProgM C C
collapse program = runSimple $ do
      (p, _, _) ← analyzeAndRewriteFwd fwd (JustC labels) program initial
      return p
  where
    labels :: [Hoopl.Label]
    labels = entryLabels program
    initial :: FactBase Fact
    initial = mapFromList (zip labels (repeat Map.empty))
    fwd :: FwdPass SimpleFuelMonad Stmt Fact
    fwd = FwdPass {fp_lattice = collapseLattice
      ,fp_transfer = collapseTransfer blockArgs
      ,fp_rewrite = collapseRewrite (destinations labels)}
    blockArgs :: Map Hoopl.Label [Var]
    blockArgs = Map.fromList [(l, args) |
                (_, BlockEntry _ l args) ← entryPoints program]
    destinations :: [Hoopl.Label] → Map Hoopl.Label DestOf
    destinations = Map.fromList . catMaybes .
                   map (uncurry destOf) . catMaybes . map (blockOfLabel program)
    destOf :: Label → Block Stmt C C → Maybe (Hoopl.Label, DestOf)
    destOf (_, l) block =
      case blockToNodeList' block of
        (JustC (CloEntry _ _ args arg), _, JustC (Done _ _ (Goto d uses))) →
          Just (l, Jump d (mapUses uses (args ++ [arg])))
        (JustC (CloEntry _ _ _ arg), _, JustC (Done _ _ (Closure d args))) →
          Just (l, Capture d (arg `elem` args))
        _ → Nothing
    mapUses :: [Name] → [Name] → [Int]
    mapUses uses args = catMaybes (map (`elemIndex` args) uses)
```

Figure 5.12: The function that puts together all definitions for our implementation of the uncurrying optimization.

The other half of Figure 5.12 describes how we create the *blocks* argument passed to *collapseRewrite*. The *destinations* function enumerates all blocks in *program* to find all closure-capturing blocks. *destOf* determines the behavior of each closure-capturing block and creates the appropriate *Jump* or *Capture* value. The result of *destinations* becomes the *blocks* argument for *collapseRewrite*.

5.8 Example: Uncurrying Across Blocks

The example shown in the previous section demonstrated that we can eliminate unnecessary `@` expressions within a block. As we will demonstrate with the next two examples, the dataflow algorithm enables us to do the same across multiple blocks, even in the presence of loops.

Uncurrying *map*

Figure 5.13 (a) shows a simple λ_C program that uses *map* to turn a list into a list of lists. Part (b) shows the MIL translation of Part (a). The listing is rather verbose as it represents the output of our λ_C to MIL compiler.

In this program, *main* applies *toList* to each element in *ns* using the *map* function. Per the definition of *map*, the *Cons* arm applies f to an element x, and then recursively calls *map* to apply f to the rest of the list. Lines 21–28 in Part (b) implement the *Cons* arm of *map*. On Line 23, `f` is applied to the element `x`. Lines 25–27 recursively call *map* with the remainder of the list. Line 28 returns the updated list.

In the body of **`cons`**, there are two opportunities to eliminate `@` expressions. `f` always represents the *toList* function, which is implemented by **`toList`** on Lines 7–11. We should be able to replace `f @ x` on Line 23 with **`toList`**`(f,x)`. Similarly,

$$\begin{aligned}
&\mathit{main}\ \mathit{ns} = \mathit{map}\ \mathit{toList}\ \mathit{ns}\\
&\mathit{map}\ f\ \mathit{xs} = \mathbf{case}\ \mathit{xs}\ \mathbf{of}\\
&\quad \mathit{Cons}\ x\ \mathit{xs}' \rightarrow \mathit{Cons}\ (f\ x)\ (\mathit{map}\ f\ \mathit{xs}')\\
&\quad \mathit{Nil} \rightarrow \mathit{Nil}\\
&\mathit{toList}\ n = \mathit{Cons}\ n\ \mathit{Nil}
\end{aligned}$$

(a)

```
main (ns):
  v227 <- k203 {}
  v228 <- k219 {}
  v229 <- v227 @ v228
  v229 @ ns
k219 {} x: toList (x)
toList (x):
  v221 <- Consclo2 {}
  v222 <- v221 @ x
  v223 <- Nil
  v222 @ v223
Consclo2 {} a2: Consclo1 {a2}
Consclo1 {a2} a1: Cons a2 a1
```

```
k203 {} f: k204 {f}
k204 {f} xs: map (xs, f)
map (xs, f):
  case xs of
    Nil -> nil ()
    Cons x xs -> cons (f, x, xs)
nil (): Nil
cons (f, x, xs):
  v209 <- Consclo2 {}
  v210 <- f @ x
  v211 <- v209 @ v210
  v212 <- k203 {}
  v213 <- v212 @ f
  v214 <- v213 @ xs
  v211 @ v214
```

(b)

Figure 5.13: A λ_C program that turns a list of elements into a list of lists and its unoptimized translation to MIL.

the recursive call to *map* can be replaced by a direct call to `map`, which implements the body of *map*.

Though our analysis covers the entire program, we first concentrate on the `main` block. Figure 5.14 shows how we analyze and rewrite `main`. The figure shows consecutive iterations of HOOPL's interleaved analysis and rewrite process. Rewrites occur between the parts of the figure; we highlight rewritten lines with a

→ symbol.

```
1   main (ns):   ⟵ {ns : ⊤}
2     v227 <- k203 {}⟵ {v227 : k203 {}}
3     v228 <- k219 {}⟵ {v228 : k219 {}}
4     v229 <- v227 @ v228⟵ {v229 : ⊤}
5     v229 @ ns
```

(a)

```
1   main (ns):
2     v227 <- k203 {}
3     v228 <- k219 {}
4 →   v229 <- k204 {v228}⟵ {v229 : {k204 {v228}}
5     v229 @ ns
```

(b)

```
1   main (ns):
2     ~~v227 <- k203 {}~~
3     v228 <- k219 {}
4     ~~v229 <- k204 {v228}~~
5 →   map (ns, v228) ⟵ {ns : ⊤},{v228 : k219 {}}
```

(c)

Figure 5.14: Development of facts and rewrites applied to the main block of our example program.

Part (a) shows the initial facts gathered about each binding in main. On Line 2, we associate v227 with the closure k203 {}. We can use this fact to rewrite the @ expression on Line 4. In Part (b), the rewritten line allows us to create a new fact, associating v229 with k204 {v228}. The closure-capturing block k204 immediately jumps to map. Therefore, on Line 5, we can rewrite the expression v229 @ ns to caseEval (ns, v228). Part (c) shows this rewrite and also crosses out lines with now-dead bindings.

After the rewrite in Figure 5.14 (c), the cfg for the program changes. `main` did not originally end in a `case` statement or "goto" expression, so the block did not have any successors; after our rewrite, `map` becomes the successor to `main`. Figure 5.15 (a) shows the cfg for our program before our rewrite; Figure 5.15 (b) shows the cfg afterwards. We also show the facts that flow between each block (using the parameters for each block to name the facts).

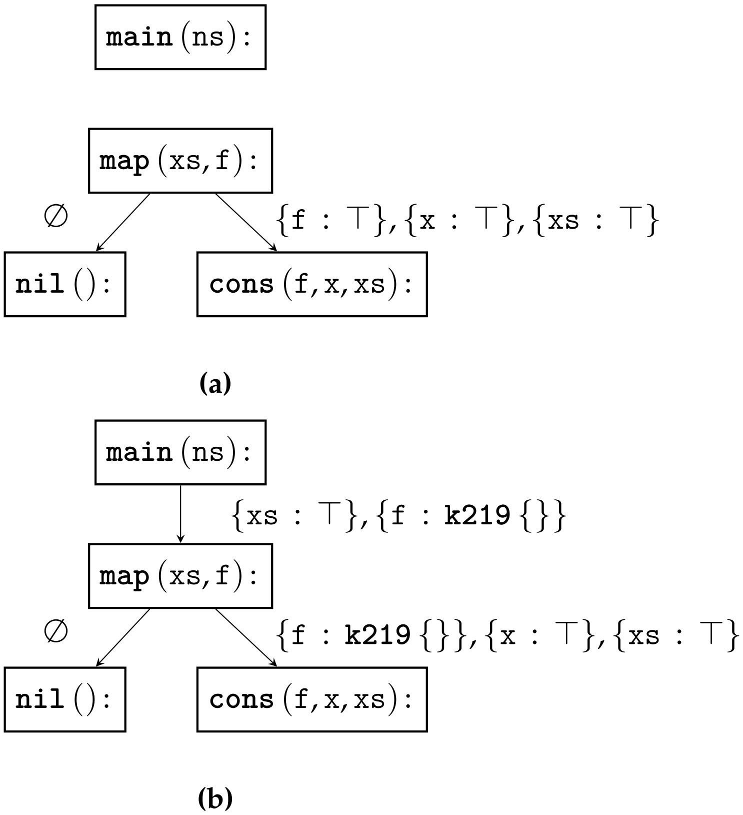


Figure 5.15: Facts that flow between blocks in our example program. Part (a) shows the cfg before we rewrite `main`; Part (b) shows the cfg afterwards. The facts from `main` only flow to the rest of the cfg after rewriting.

As Figure 5.15 (b) shows, when **map** becomes the successor of **main**, the fact {f : **k219**{}} becomes available to **cons**. Figure 5.16 shows how we iteratively analyze and rewrite **cons** using our new fact. Part (a) shows the initial facts for each binding. In Part (b), we replace the expression f @ x on Line 3 with **toList**(x), because we know the closure-capturing block **k219** immediately jumps to **toList**. On Line 6, we rewrite v212 @ f to **k204**{f}, due to the fact {v212 : **k203**{}} and that **k203** returns **k204**{f}. Originally, this line gathered the first argument for *map*; now, we create the closure directly. This also generates a new fact, {v213 : **k204**{f}}. We know that **k204** jumps to **map** (i.e., the body of *map*). In Part (c), we use our knowledge of v213 to rewrite Line 7 from v213 @ xs to **map**(xs, f). We also cross out dead bindings that could be eliminated, after our rewrites.

Figure 5.17 summarizes the result of applying our uncurrying optimization (and dead-code elimination) to the program in Figure 5.13. On Line 9, we replaced the expression f @ x with **toList**(x); our program now directly calls *toList*, rather than repeatedly entering the closure represented by f. In Figure 5.13, Lines 25–27 implemented the recursive call to *map*. In Figure 5.17, Line 11 replaces those three lines with **map**(xs, f), a direct recursive call. The first change does not save a closure allocation (because f is still passed in),[5] but the second change saves two closure allocations and two @ expressions.

[5] We could eliminate f through an analysis that finds unused parameters.

```
cons (f, x, xs): ⟵ {f : k219 {}}, {x : ⊤}, {xs : ⊤}
  v209 <- Consclo2 {} ⟵ {v209 : Consclo2 {}}
  v210 <- f @ x ⟵ {v210 : ⊤}
  v211 <- v209 @ v210 ⟵ {v211 : ⊤}
  v212 <- k203 {} ⟵ {v212 : k203 {}}
  v213 <- v212 @ f ⟵ {v213 : ⊤}
  v214 <- v213 @ xs ⟵ {v214 : ⊤}
  v211 @ v214
```

(a)

```
cons (f, x, xs):
  v209  <- Consclo2 {}
→v210 <- toList (x) ⟵ {v210 : ⊤}
→v211 <- Consclo1 {v210} ⟵ {v211 : Consclo1 {v210}}
  v212 <- k203 {}
→v213 <- k204 {f} ⟵ {v213 : k204 {f}}
  v214 <- v213 @ xs
  v211 @ v214
```

(b)

```
cons (f, x, xs):
  ~~v209 <- Consclo2 {}~~
  v210 <- toList (x)
  v211 <- Consclo1 {v210}
  ~~v212 <- k203 {}~~
  ~~v213 <- k204 {f}~~
→v214 <- map (xs, f) ⟵ {v214 : ⊤}
  v211 @ v214
```

(c)

Figure 5.16: Development of facts and rewrites within `cons`, after facts begin flowing from `main`.

```
main(ns):
  v228 <- k219{}
  map(ns, v228)
k219{} x: toList(x)
toList(x):
  v222 <- Consclo1{x}
  v223 <- Nil
  v222 @ v223
Consclo1{a2} a1: Cons a2 a1
```

```
k203{} f: k204{f}
k204{f} xs: map(xs, f)
map(xs, f):
  case xs of
    Nil -> nil()
    Cons x xs -> cons(f, x, xs)
nil(): Nil
cons(f, x, xs):
  v210 <- toList(x)
  v211 <- Consclo1{v210}
  v214 <- map(xs, f)
  v211 @ v214
```

Figure 5.17: Our MIL program from Figure 5.13 after applying our uncurrying optimization. We also removed unused blocks and unnecessary bindings within blocks.

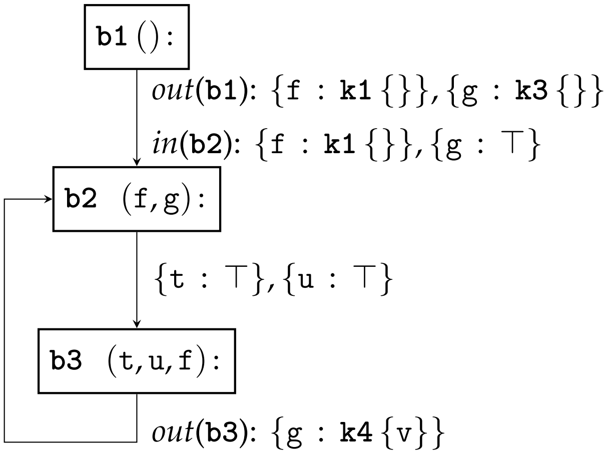
Uncurrying Across Loops

Our next example demonstrates uncurrying in the presence of loops. Figure 5.18 gives our example MIL program and its CFG; the program itself does not do anything very interesting, but we are concerned with its structure rather than its behavior. Note that we only show the normal blocks (`b1`, `b2`, and `b3`) in the CFG, as the control-flow between each pair of closure-capturing blocks is not very relevant.

We annotated the CFG in Figure 5.18 (b) with the initial facts between each block. Recall that in a forwards dataflow analysis, the *in* facts for a block are computed using the meet of *out* facts from predecessor blocks. As `b2` has two predecessors, we explicitly show the *out* facts for `b1` and `b3`. Notice that *out*(`b3)` does not contain a fact for `f`; because no binding to `f` occurs in `b3`, no fact will (yet) appear in *out*(`b3)`. In turn, this means *in*(`b2`) contains the fact {`f : k1{}`} from *out*(`b1`). In `b3`, the statement `w <- k4 {v}` ultimately creates the fact {`g : k3 {}`} in *out*(`b3`). However, *out*(`b1`) contains {`g : k3 {}`}. Because these values differ, *in*(`b2`) contains the fact {`g : ⊤`}.

```
1  b1 ():
2    f <- k1 {}
3    g <- k3 {}
4    b2 (f, g)
5  b2 (f, g):
6    t <- f @ g
7    u <- g @ t
8    b3 (t, u, f)
9  b3 (t, u, f):
10   v <- f @ t
11   w <- k4 {v}
12   b2 (f, w)
13 k1 {} x: k2 {x}
14 k2 {x} y: Left y
15 k3 {} x: k4 {x}
16 k4 {x} y: Right y
```

(a)



(b)

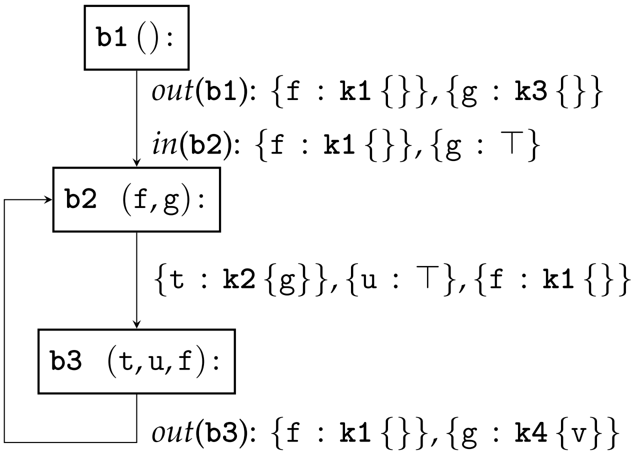
Figure 5.18: A MIL program with looping control-flow.

The initial facts in Figure 5.18 tell us that `f` refers to the closure-capturing block `k1`, which lets us replace the expression `f @ g` on Line 6 with `k2 {g}`. Similarly, the same fact propagates to `b3`, allowing us to rewrite the expression `f @ t` on Line 10 to `k2 {t}`.

Figure 5.19 shows the rewritten program and updated facts. After these rewrites, the fact sets reach a fixed point and the analysis stops. Applications of `f` are correctly replaced with direct closure allocations, but applications of g remain

```
b1():
  f <- k1{}
  g <- k3{}
  b2(f, g)
b2 (f, g):
  t <- k2{g}
  u <- g @ t
  b3(t, u, f)
b3 (t, u, f):
  v <- k2{t}
  w <- k4{v}
  b2(f, w)
k1{} x: k2{x}
k2{x} y: Left y
k3{} x: k4{x}
k4{x} y: Right y
```

(a)


b1():
out(b1): {f : k1{}},{g : k3{}}
in(b2): {f : k1{}},{g : ⊤}
b2 (f,g):
{t : k2{g}},{u : ⊤},{f : k1{}}
b3 (t,u,f):
out(b3): {f : k1{}},{g : k4{v}}

(b)

Figure 5.19: Our rewritten MIL program, showing that we correctly uncurried `f @ g` in `b2`; `g @ t` remains unchanged.

as it does not always hold the same closure.

5.8.1 Soundness

Our implementation of uncurrying can produce incorrect results under two circumstances. In the first case, we can introduce free variables into a block. In the second case, our analysis does not see facts that should be propagated to a block, leading to unsound rewrites. We describe both cases, and possible solutions, below.

The first case occurs when a function application is replaced with a closure that introduces variables not declared in the containing block. When *collapseTransfer* sees a binding to a closure value, it records not only the label that the closure refers to, but also all of the variables captured in the closure. These facts are propagated to successor blocks. If those blocks are subsequently rewritten to allocate the closure directly, then the variables in the closure may be "unpacked" into the block, introducing free variables that are not properly bound.

For example, consider the MIL program in Figure 5.20. In Part (a), the statement `v <- k1 {x}` in **b1** binds v to **k1** {x}. The closure is then passed to **b2**, which applies the closure to y and returns the result.

```
b1 (x, y):             b1 (x, y):
  v <- k1 {x}            v <- k1 {x}
  b2 (v, y)              b2 (v, y)

b2 (v, y):             b2 (v, y):
  t <- v @ y             t <- p1 (x, y)
  return t               return t
k1 {x} y: p1 (x, y)    k1 {x} y: p1 (x, y)
p1 (x,y): ...          p1 (x,y): ...

        (a)                    (b)
```

Figure 5.20: A MIL program that demonstrates how free variables can be accidentally introduced by uncurrying. Part 5.20 (a) shows the original program. In Part 5.20 (b), rewriting **b2** introduced the free variable x.

Our analysis create the fact {v,**k1** {x}} when analyzing **b1**, which then propagates to **b2**. In **b2**, we would rewrite the expression `v @ y` to **p1** (x,y), as **k1** immediately jumps to **p1**, producing the program shown in Part 5.20 (b). But this introduces a free variable, x, in **b2**.

This problem might be solved with another dataflow analysis. After uncurrying, we would determine the free variables in each block (a backwards dataflow analysis). Our uncurrying analysis could keep track of where each variable in a given closure was declared. We could use that information to propagate free variables from the block in which they are first bound to the blocks where they are used.

The second case occurs when a block is called on the right-hand side of a bind statement, such as `v <- b (...)`. Our analysis will not propagate any facts to `b` in such situations. If the values passed to the block `b` on the right-hand side of a `<-` differ from those passed at the end of a block, then our analysis may rewrite using partial facts.

```
1  b1 (x):
2    v <- k1 {}
3    w <- k2 {}
4    z <- b2 (w, y)
5    b2 (v, z)

6  b2 (v, y):
7    t <- v @ y
8    return t
9  k1 {} x: Left x
10 k2 {} x: Right x
```

(a)

```
1  b1 (x):
2    v <- k1 {}
3    w <- k2 {}
4    z <- b2 (w, y)
5    b2 (v, z)

6  b2 (v, y):
7    t <- k1 {}
8    return t
9  k1 {} x: Left x
10 k2 {} x: Right x
```

(b)

Figure 5.21: A mil program demonstrating problems with "call" expressions on the right-hand side of a bind.

Figure 5.21 demonstrates this issue. Block `b1` allocates two closures, `v` to `k1 {}` and `w` to `k2 {}`. On Line 4, the program calls `b2` with `w`; Line 5 calls `b2` with `v`. Our analysis would only consider the second call to `b2` and would deduce that `v` is always `k1` {} in `b2`. Figure 5.21 (b) shows the rewritten program. In `b2`, `t <- v @ y` has been incorrectly rewritten to `t <- k1` {}, which is incorrect.

A simple solution to this problem would first scan the entire program, finding all blocks called on the right-hand side of a bind; then those blocks would be eliminated from further analysis. Our intuition is that while this solution is not ideal, many programs can still be uncurried even with this restriction.

5.9 Related Work

Appel (1992, Section 6.2) describes uncurrying in the context of a compiler that uses continuation-passing style, though CPS conversion is not essential to the transformation. While we described uncurrying in terms of one-argument functions, Appel allows tuples of arguments. His approach looks for functions whose bodies only apply a locally defined, non-recursive function. Using our λ_C notation with tuples, this pattern looks like:

$$\begin{array}{l} f\ (x, c) = \\ \quad \textbf{let}\ g\ (y, k) = E \\ \quad \textbf{in}\ c\ g \end{array}$$

where E represents the non-recursive body of g.

To uncurry f, Appel creates a new function, f', that takes all arguments to f and g at once. He then rewrites f to use f':

$$\begin{array}{l} f'\ (x, c, g, y, k) = E \\ f\ (x, c) = \\ \quad \textbf{let}\ g\ (y, k) = f'\ (x, c, g, y, k) \\ \quad \textbf{in}\ c\ g \end{array}$$

This transformation preserves the original f, as not all call sites of f may be known. Known call sites, however, can use the more efficient version, f'. Appel describes what makes f' efficient and how other optimizations can remove the unused arguments in f' and f.

Like our version of uncurrying, Appel relies on other optimizations to clean up. Unlike our version, Appel's looks for a specific syntactic form to transform. Our use of dataflow analysis allows us to rewrite any function application that we can prove always uses the same closure. Appel's version appears to only apply to a very specific form of curried definitions, most of which are produced by the translation to CPS.

Tarditi (1996) describes an uncurrying optimization that extends Appel's work. In fact, Tarditi points out that Appel's description is only guaranteed to work for functions of two arguments; for more arguments, Appel's transformation must be applied in a specific order (which Appel did not describe).

Tarditi's approach uses four passes to uncurry functions of the form $(\lambda x.\ \lambda.\ y.\ \ldots)\ a\ b$ into $(\lambda(x,y).\ \ldots)\ (a,b)$, where tuples represent a multi-argument function. The first pass of Tarditi's algorithm scans all definitions in the program to find non-recursive, curried definitions, and records their arity (i.e., the number of nested λs). The second pass looks for applications of curried functions to arguments. He again scans the program, searching for specific declarations that partially apply a curried function. He is also able to recognize subsequent applications of previous partial applications, extending the number of arguments associated with a given sequence of applications. The third pass of his algorithm creates new, uncurried definitions of the curried functions found in the first pass.

The fourth and final pass of the algorithm converts those applications found in the second pass that are fully applied to use the new, uncurried versions of the curried function they originally referred to.

Tarditi's algorithm is not limited to finding curried functions of a certain syntactic form, and it extends correctly to functions of multiple arguments. His algorithm, however, only replaces fully-applied functions. Our analysis can replace any candidate function application, even if it does not result in a fully applied function.

Tolmach and Oliva (1998) do not specifically describe an uncurrying optimization; rather, they describe how "closure conversion," plus two other general optimizations, give them uncurrying for free in their compiler for Standard ML. Critically, their compiler uses closure-conversion to remove all higher-order functions from the program and replace them with functions that return a data structure representing the original closure. Applications of curried functions are replaced with calls to a dispatching function that uses case discrimination to distinguish closures of the same arity, calling the uncurried version of each curried function.

Tolmach and Oliva's compiler uses inlining and "case splitting" to ensure the program does not trade the cost of a partial application for the cost of a data allocation and a call to the dispatching function. Any application of the curried function will be inlined into the call site, as the body of the curried function just allocates a data structure. The call to the dispatch function will now use the data structure inlined from the curried function. The dispatch function only contains a case statement that discriminates based on the data structure representing each curried function. "Case splitting" replaces the entire call site with the relevant arm from the dispatch function, thus turning an allocation, case discrimination and function call into just a function call.

Tolmach and Oliva's approach does not depend on recognizing syntactic patterns at all. It should recognize any known call site of a partially applied function, and they claim it works for functions of multiple arguments as well. Our approach is similar, in that we look for bindings known to refer to closure values. We even use a simple form of inlining, meaning we inspect the *tail* found in the closure-capturing block referred to by a given closure and "inline" the tail when it is a direct jump or closure allocation. Our use of dataflow analysis, however, distinguishes our work, in that we do not depend on function applications to take on a particular form. Once we determine that the left-hand side of a given `@` expression always refers to the same closure, we can transform the expression by a simple rewrite using the body of the closure-capturing block.

5.10 Conclusion

In this chapter, we described an uncurrying optimization for MIL programs in terms of the dataflow algorithm. We gave dataflow equations detailing the optimization, setting our algorithm on a solid theoretical foundation. We implemented our algorithm using the HOOPL library, and gave a complete and detailed presentation of that work. By example, we demonstrated the utility of our optimization. We discussed challenges in our current implementation, and offered suggestions for improving the algorithm in the future. Finally, we compared our implementation to several other implementations of the uncurrying optimization in the literature.

Chapter 6

Conclusion & Future Work

Our work applied the dataflow algorithm to an area outside its traditional scope: functional languages. We based our work on a *monadic intermediate language* (MIL) that supported high-level functional programming and exposed certain low-level implementation details. We implemented our analysis in Haskell using the HOOPL library; we also gave a thorough description of how to implement dataflow-based optimizations using HOOPL. We then demonstrated the utility of our work by using HOOPL and MIL to create a novel implementation of the uncurrying optimization.

Section 6.1 describes several optimizations we developed that are based on the monadic properties of MIL, rather than dataflow analysis. We discuss a number of extensions to our work in Section 6.2. Section 6.3 describes challenges we encountered using the HOOPL library, and gives some suggestions for improvements. Section 6.4 offers our closing thoughts.

6.1 Monadic Optimizations

While this work focuses on MIL, HOOPL, and our uncurrying implementation, we also developed several optimizations that relied on the monadic properties of MIL. In Section 6.1.1 we describe an inlining optimization based on the *monad laws*. Section 6.1.2 describes how we can safely eliminate dead-code, again using the monadic properties of MIL.

6.1.1 Inlining Monadic Code

Figure 6.1 shows the *monad laws*: Left-Unit, Right-Unit, and Associativity. While these laws can be interpreted as specifications of behavior, they can also be interpreted as *transformations*.

$$\begin{aligned}
\textbf{do}\ \{x \leftarrow \mathit{return}\ y; m\} &\equiv \textbf{do}\ \{[y \mapsto x]\ m\} && \mathit{Left\text{-}Unit} \quad (6.1)\\
\textbf{do}\ \{x \leftarrow m; \mathit{return}\ x\} &\equiv \textbf{do}\ \{m\} && \mathit{Right\text{-}Unit} \quad (6.2)\\
\textbf{do}\ \{x \leftarrow \textbf{do}\ \{y \leftarrow m; n\}; o\} &\equiv \textbf{do}\ \{y \leftarrow m; x \leftarrow n; o\} && \mathit{Associativity} \quad (6.3)
\end{aligned}$$

Figure 6.1: The monad laws, as stated by Wadler (1995). The notation "$[y \mapsto x]\ m$" means that y should be substituted for x everywhere in m.

For example, the following block binds x to the value of y, keeping both variables live between the "x <- **return** y" and l(x, y) statements:

```
b():
  x <- return y
  ...
  l(x, y)
```

If no intervening statement binds x again, we can use the Left-Unit law to replace all occurrences of x with y:

```
b():
  x <- return y
  ...
  l(y, y)
```

Because we know **return** y produces no side-effects, we can eliminate the binding for x. If variables represent registers, this optimization reduces the number of registers used by the block and makes it smaller:

```
b():
  ...
  l(y, y)
```

The Right-Unit law shortens the "tail" of MIL blocks that end with a **return** statement. For example, Right-Unit can be used to transform the following block:

```
b(...):
  ...
  x <- f @ y
  return x
```

into the shorter block:

```
b(...):
  ...
  f @ y
```

Not only does this transformation eliminate a redundant **return** statement, it may also allow further optimizations. In particular, if we know that the closure represented by f refers to block b, our uncurrying optimization will transform f @ y into either a jump or an allocation.

The Associativity law provides an inlining mechanism for MIL programs. The inner monadic computation mentioned on the right-hand side of the law, $\mathbf{do}\ \{y \leftarrow m; n\}$, can be an arbitrarily long monadic program. All MIL blocks are monadic programs — therefore, we can use this law to inline almost any block. For example, consider these two blocks:

```
compose(f, g, x):
  t1 <- g @ x
  t2 <- f @ t1
  return t2

main(a,b,c):
  x <- compose(a,b,c)
  b(x)
```

Equation (6.3) lets us inline `compose` into `main`, as long as we appropriately rename variables:

```
main(a,b,c):
  t1 <- b @ c
  t2 <- a @ t1
  x <- return t2
  b(x)
```

Notice that we can now also apply Equation (6.1), eliminating the use of `x`:

```
main(a,b,c):
  t1 <- b @ c
  t2 <- a @ t1
  b(t2)
```

Mil's syntax does not allow all monadic blocks to be inlined. Mil only allows branching at the end of a block; therefore, blocks that end in `case` statements cannot be inlined.

However, we can still transform around blocks that end in `case` statements. Note that we did not implement this particular form of inlining (though we did implement that given above). Consider the blocks `b1`, `t` and `f` in the following program:

```
b1(a):
   t1 <- b2(a)
   ...
   b3(t1)

b2(a):
  case a of
    True -> t(a)
    False -> f(a)

t(...):
   ...
   return x

f(...):
   ...
   return x
```

`b1` binds t1 to the result of `b2`. `b2` returns the result of either block `t` or `f`. Because `t` and `f` do not end in a `case` statement, we can move the code that follows t1's binding in `b1` to `t` and `f`:

```
b1(a):
   b2(a)

b2(a):
  case a of
    True -> t(a)
    False -> f(a)

t(a):
   ...
   t1 <- return x
   b3(t1)

f(...):
   ...
   t1 <- return x
   b3(t1)
```

This new program may be more efficient. For example, blocks `t` and `f` end in tail calls, where before they ended in a `return`. We can use the Left-Unit law to substitute `x` for `t1` in `t` and `f` as well (which, in turn, allows us to remove `t1`'s binding in both blocks as it is no longer live). We may be able to rewrite calls `b1` to `b2`, and remove `b1` altogether. Finally, the "Push Through Cases" optimization described in Section 6.2.2 may be able to optimize `t` and `f` even further.

6.1.2 Dead-Code Elimination

Mil treats allocation as a monadic operation, and allocation can definitely cause observable effects. However, most of the time we do not mind eliminating those effects, as they only cause our program to behave badly. Therefore it is usually reasonable to remove any allocation (be it a closure, thunk or constructor) that binds to a dead variable.

For example, consider *compose1*, which captures the first argument to *compose*:

$$compose1\ f = compose\ f\ .$$

And the corresponding mil code:

```
compose1 (): absBodyL208 {}
absBodyL208 {} f: absBlockL209 (f)
absBlockL209 (f):
  v210 <- compose ()
  v210 @ f

compose (): absBodyL201 {}
absBodyL201 {} f: absBodyL202 {f}
absBodyL202 {f} g: ...
```

We can use the Associativity law to inline the allocation returned by `compose` into `absBlockL209`, giving us:

```
absBlockL209(f):
  v210 <- absBodyL201{}
  v210 @ f
```

Our uncurrying optimization can determine that the expression `v210 @ f` evaluates to `absBodyL202{f}` (because `v210` holds the closure `absBodyL201{}`). We can replace `v210 @ f` with `absBodyL202{f}`:

```
absBlockL209(f):
  v210 <- absBodyL201{}
  absBodyL202{f}
```

After this rewrite, `v210` is no longer live. Because closure allocation has no observable side-effect, we remove the binding, eliminating one allocation:

```
absBlockL209(f):
  absBodyL202{f}
```

This optimization can be extended to thunk and data allocations.

6.2 Future Work

We discuss how to extend our uncurrying optimization to thunks in Section 6.2.1. Section 6.2.2 proposes a new analysis to eliminate unnecessary allocations across `case` statements.

6.2.1 Eliminating Thunks

Monadic thunks and closures share many characteristics. For example, they both represent suspended computation, and they both capture an environment of values. They also can be a source of inefficiency, as well. Our compiler for λ_C to MIL produces many blocks that immediately invoke some thunk. For example, the following λ_C definition reads a character and prints it to the screen:

```
main x = do
  c ← readChar
  print c
```

Our compiler translates the program in this MIL code (in part):

```
main ():
  v206 <- readCharbody []
  c <- invoke v206
  ...
readCharbody (): readChar*()
```

In this program, `main` allocates a thunk pointing to `readCharbody` and binds it to `v206`. The next line invokes the thunk just constructed, binding the result to `c`. A straightforward adaption of our uncurrying optimization could transform this program so it executes `readCharbody` directly, instead of invoking the thunk:

```
main ():
  v206 <- readCharbody []
  c <- readCharbody ()
  ...
```

Of course, we can continue to apply further optimizations to the program. Dead-code elimination would find that `v206` is no longer live, letting us eliminate the allocation of the thunk:

```
main ():
  c <- readCharbody ()
  ...
```

The associativity law also lets inline the body of **readCharBody** into **main**, removing an extra jump:

```
main():
  c <- readChar*()
  ...
```

While these last transformations do not relate directly to eliminating thunks, they do show that each optimization tends to make further optimizations possible.

6.2.2 Push Through Cases

Functional language programs commonly implement a pattern of *construct/destruct*, where the program constructs a value and then inspects (or destructs) the value shortly thereafter. Figure 6.2 shows one such program. The *dec* function returns a *Maybe* value, indicating if its argument could be decremented or not. The *loop* function discriminates on the result of *dec n*, immediately throwing away the *Maybe* value created by *dec*. The "safe" decrement implemented by *dec* guarantees we will not apply f to values less than 0.

Figure 6.3 shows unoptimized MIL code for these two functions. **loop** evaluates **dec** (n) on Line 2 and binds the result to v215. The **case** statement on the next line immediately takes v215 apart, throwing away the allocated value just created.

Inspecting the **dec** block in Figure 6.3 shows that it evaluates a condition and branches to either **altTrue** or **altFalse**. As discussed in Section 6.1.1, we cannot directly inline **loop** into **dec**, because **loop** ends with a **case** statement. However, we can move the body of **loop** into each arm of the **case** statement that ends **loop**.

We begin by inlining **dec** into **loop**. Notice that the **case** statement now jumps to **altTrue** and **altFalse**, where before it jumped to **altJust** and **altNothing**:

```
dec :: Int → Maybe Int
dec i = if i > 0
  then Just (i − 1)
  else Nothing

loop :: Int → (Int → Int) → Int
loop n f = case dec n of
  Just i → loop (f i) f
  Nothing → f 0
```

Figure 6.2: A program that illustrates the *construct/destruct* pattern.

```
loop(n, f):
  v233 <- gt*(i, 0)
  case v233 of
    True -> altTrue(i, f, n)
    False -> altFalse(f, n)
```

We also move the original case statement from `loop` to the end of `altTrue` and `altFalse`. This transformation requires that we bind the original result of `altTrue` and `altFalse` to the variable that the original case statement inspected (`v215`). For example, `altTrue` previously returned `Just *(v225)`; now, we bind `v215` to that value. In both blocks, the value bound is immediately destructed by a `case` statement:

```
altTrue(i, f, n):
  v225 <- minus*(i, 1)
  v215 <- Just*(v225)
  case v215 of
    Just i -> altJust(f, i)
    Nothing -> altNothing(f)
```

```
loop(n, f):
  v215 <- dec(n)
  case v215 of
    Just i -> altJust(f, i)
    Nothing -> altNothing(f)

dec(i):
  v233 <- gt*(i, 0)
  case v233 of
    True -> altTrue(i)
    False -> altFalse()

altNothing(f): f @ 0

altJust(f, n):
  v207 <- f @ n
  loop(v207, f)

altTrue(i):
  v225 <- minus*(i, 1)
  Just*(v225)

altFalse(): Nothing*()
```

Figure 6.3: Initial form of our function.

```
altFalse(f, n):
  v215 <- Nothing*()
  case v215 of
    Just i -> altJust(f, n)
    Nothing -> altNothing(f)
```

Dataflow analysis of `altTrue` and `altFalse` could show that each block contains a case alternative that will never be executed. For example, in `altTrue`, `v215` must always be a `Just` value, and the `Nothing` alternative will never execute. We can eliminate the case statement in both blocks and replace them with a jump.

Notice that, in the `altTrue` case, we need to recognize that `i` in `Justi` is really `v225`:

```
altTrue(i, f, n):
  v225 <- minus*(i, 1)
  v215 <- Just*(v225)
  altJust(f, v225)

altFalse(f, n):
  v215 <- Nothing*()
  altNothing(f)
```

Dead-code elimination would find that the bindings for `v215` in both blocks is dead, and would eliminate the allocation. Figure 6.4 shows the final form of our program, where we have eliminated the unnecessary allocation between `dec` and `loop`. This version of the program will perform no allocations of *Maybe* values whatsoever, but we are still guaranteed that *f* will not be applied to an index value less than 0.

6.3 Hoopl Refinements

The HOOPL library played a critical role in our work. The library presents a simple and powerful interface for describing, analyzing and transforming CFGS. HOOPL allowed us to explore a variety of optimizations that used dataflow analysis, without the burden of implementing the dataflow algorithm from scratch. Of course, by spending so much time with the library, we found some areas where the library's interface left us struggling to bend our algorithm to fit with HOOPL's view of dataflow analysis. The following sections describe the issues we encountered, and offer some possible solutions.

```
loop(n, f):
   v233 <- gt*(i, 0)
   case v233 of
      True -> altTrue(i, f, n)
      False -> altFalse(f, n)

altTrue(i, f, n):
   v225 <- minus*(i, 1)
   altJust(f, v225)

altFalse(f, n):
   altNothing(f)

altNothing(f): f @ 0

altJust(f, n):
   v207 <- f @ n
   loop(v207, f)
```

Figure 6.4: Final form of our function.

6.3.1 Invasive Types

Hoopl uses the *O* and *C* types (described on Page 31) to specify the shape of each node in a cfg; only nodes with compatible types can be connected to each other. This design allows the compiler to enforce some desirable properties; for example, a basic block will not contain any nodes that can branch to more than one destination. Unfortunately, this design also requires that the *O* and *C* types be present on the client ast. In preliminary work, we implemented an ast without using hoopl's shape types. This choice required us to write a significant amount of boilerplate to translate between our initial representation and one that used hoopl's desired types.

"Smart" constructors could be used to reduce the boilerplate required when using HOOPL against an existing AST. For example, consider the the AST given in Figure 3.3 on Page 33. Instead of defining *CStmt* using GADTS, imagine we defined *CStmtX* as a normal ADT and *CStmt* as a **newtype**:

```
data CStmtX = Entry Label |
  Assign Var CExpr
  ...
newtype CStmt o c = CStmt CStmtX
```

To create HOOPL-ized values, we define a function for each *CStmtX* constructor, parameterized by shape:

```
entry :: Label → CStmt O C
entry l = CStmt (Entry l)
assign :: Var → CExpr → CStmt O O
assign v e = CStmt (Assign v e)
...
```

However, this approach still requires a fair amount of boilerplate code. GADTS alleviate the problem somewhat (since the compiler implements "smart" constructors for you), but that does not help when working against an AST that cannot be changed. Metaprogramming techniques using Template Haskell may ultimately be the best approach here.

6.3.2 Restricted Signatures

HOOPL does not specify transfer and rewrite functions using simple function signatures. Instead, as detailed in Section 3.4 (Page 39), HOOPL represents those functions using the *BwdTransfer*, *BwdRewrite*, *FwdTransfer* and *FwdRewrite* types.

Client programs cannot directly create these values; instead, HOOPL defines a function for creating each type:

```
mkFTransfer :: (forall e x . n e x → f → Fact x f) → FwdTransfer n f
mkBTransfer :: (forall e x . n e x → Fact x f → f) → BwdTransfer n f
mkFRewrite :: FuelMonad m ⇒
  (forall e x . n e x → f → m (Maybe (Graph n e x)))
   → FwdRewrite m n f
mkBRewrite :: FuelMonad m ⇒
  (forall e x . n e x → Fact x f → m (Maybe (Graph n e x)))
   → BwdRewrite m n f
```

Unfortunately, this scheme complicated our implementation at times. Hoopl's type for transfer functions only allows information to be stored in three places: the client's AST, the facts computed, and any values declared in some scope external to the transfer function. Each of these locations leads to different complications.

To illustrate, imagine a forwards transfer function that analyzes a block statement by statement (so it does not have access to an entire block at once), but that also needs to know the label of the current block being analyzed. Hoopl requires that such a function have the signature:

```
forall e x . n e x → f → Fact x f
```

We could store the label of the current block in the client's AST; that would mean each value of type *n* would need to hold a label representing the current block. Possible, but burdensome at least. The label of the current block could be part of the facts computed. This works, but seems wasteful, as the label would not matter outside each block, but it would be carried by all values of type *f*. We could also capture the current label for the block in some outer scope, but that seems to imply we would be applying hoopl to a single block at a time, which would not work for any inter-block analysis.

A simple accumulating parameter on transfer function would alleviate this issue. Hoopl does allow the client program to define a custom monad that will be used by the rewrite functions, and that can give access to intermediate results. Unfortunately, the custom monad still does not help with the transfer function.

Earlier versions of the hoopl library allowed the client to return an arbitrary *function* from the transfer and rewrite functions. While that may have been too liberal, we certainly wished for a slightly less restricted interface during our work.

6.4 Summary

Kildall applied his dataflow algorithm to algol 60, an imperative, structured programming language. Most work in dataflow analysis since then has focused on imperative programming languages. We set out to explore the algorithm's use within the context of a functional programming language; specifically, we hypothesized that, by compiling to a monadic intermediate language, we could obtain a basic-block structure that would be amenable to dataflow analysis. We intended to implement optimizations drawn from the literature of imperative and functional compilers, showing that the algorithm could be applied in both contexts.

Mil builds on a large body of work on monadic programming, intermediate languages, and implementation techniques for functional languages. While a monadic intermediate language is not new, we believe mil's combination of low-level and high-level language features makes it unique. Mil exposes the allocation of closures and other implementation details, but still offers high-level features like function application and case discrimination. Mil programs, by design, consist of basic-block elements. Of course, many intermediate languages consist of basic blocks, but MIL combines that structure with a monadic programming model, giving a "pure" flavor to low-level operations.

Our work made significant use of the HOOPL library. Without it, we may not have even pursued this research. While we did not contribute materially to HOOPL itself, this work offers a significant amount of expository material describing HOOPL, as well as at least one implementation of a non-trivial optimization that cannot be found elsewhere.

Finally, our work described a novel implementation of uncurrying, based on dataflow analysis. We showed that our optimization works across multiple blocks and in the presence of loops. We also were able to combine uncurrying with optimizations based on monadic transformations, though were not able to describe those as fully here.

This work provides a complete and accurate description of our MIL language, our uncurrying optimization, and its implementation in HOOPL. We hope that future readers use our work to implement their own dataflow-based analysis, see a standalone example of HOOPL in a non-trivial setting, or even implement a MIL of their own.

Bibliography

Aho, Alfred V. et al. (2006). *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Boston, MA: Addison-Wesley Longman Publishing Co., Inc.

Appel, Andrew W. (1992). *Compiling with Continuations*. New York, NY, USA: Cambridge University Press.

Benton, Nick, Andrew Kennedy, and George Russell (1998). "Compiling Standard ML to Java Bytecodes". In: *Proceedings of the Third ACM SIGPLAN International Conference on Functional Programming*. ICFP '98. Baltimore, Maryland, United States: ACM, pp. 129–140.

Benton, Nick, Andrew Kennedy, and Claudio V. Russo (2004). "Adventures in Interoperability: The SML.NET Experience". In: *Proceedings of the 6th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming*. PPDP '04. Verona, Italy: ACM, pp. 215–226.

Benton, Nick et al. (2005). "Shrinking Reductions in SML.NET". In: *Implementation and Application of Functional Languages*. Ed. by Clemens Grelck et al. Vol. 3474. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, pp. 1–9.

Flanagan, Cormac et al. (1993). "The Essence of Compiling with Continuations". In: *Proceedings of the ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation*. PLDI '93. Albuquerque, New Mexico: ACM, pp. 237–247.

HASP (2010a). *A Habit Compilation Strategy*. Available privately; contact Mark P. Jones at Portland State University.

— (2010b). *The Habit Programming Language: The Revised Preliminary Report*. URL: `http://hasp.cs.pdx.edu/habit-report-Nov2010.pdf`.

Kennedy, Andrew (2007). "Compiling with Continuations, Continued". In: *Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming*. ICFP '07. Freiburg, Germany: ACM, pp. 177–190.

Kildall, Gary A. (1973). "A Unified Approach to Global Program Optimization". In: *Proceedings of the 1st Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*. POPL '73. Boston, Massachusetts: ACM, pp. 194–206.

Lerner, Sorin, David Grove, and Craig Chambers (2002). "Composing Dataflow Analyses and Transformations". In: *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '02. Portland, Oregon: ACM, pp. 270–282.

Moggi, Eugenio (1991). "Notions of Computation and Monads". In: *Information and Computation* 93 (1), pp. 55–92.

Ramsey, Norman, João Dias, and Simon Peyton Jones (2010). "Hoopl: a Modular, Reusable Library for Dataflow Analysis and Transformation". In: *Proceedings of the Third ACM Haskell Symposium on Haskell*. Haskell '10. Baltimore, Maryland: ACM, pp. 121–134.

— (Mar. 2011). *HOOPL*. Version 3.8.7.0. URL: `http://hackage.haskell.org/package/hoopl-3.8.7.0`.

Schrijvers, Tom et al. (2009). "Complete and Decidable Type Inference for GADTs". In: *Proceedings of the 14th ACM SIGPLAN International Conference on Functional Programming*. ICFP '09. Edinburgh, Scotland: ACM, pp. 341–352.

Tarditi, David (1996). "Design and Implementation of Code Optimizations for a Type-Directed Compiler for Standard ML". PhD thesis.

The GHC Team (2011a). *GHC 7.2 User's Manual*. URL: `http://www.haskell.org/ghc/docs/7.2.2/html/users_guide`.

— (2011b). *The Glasgow Haskell Compiler*. URL: `http://www.haskell.org/ghc/`.

Tolmach, Andrew and Dino P. Oliva (1998). "From ML to Ada: Strongly-typed Language Interoperability via Source Translation". In: *J. Funct. Program.* 8 (4), pp. 367–412.

Wadler, Philip (1990). "Comprehending Monads". In: *Proceedings of the 1990 ACM Conference on LISP and Functional Programming*. LFP '90. Nice, France: ACM, pp. 61–78.

— (1995). "Monads for Functional Programming". In: *Advanced Functional Programming, First International Spring School on Advanced Functional Programming Techniques-Tutorial Text*. London, UK: Springer-Verlag, pp. 24–52.

Whalley, David B. (1994). "Automatic Isolation of Compiler Errors". In: *ACM Transactions on Programming Languages and Systems* 16 (5), pp. 1648–1659.

Appendix

Source Code

The source code for the λ_C to MIL compiler, our uncurrying optimization, the dead-code elimination example from Chapter 3, the "monadic" optimizations mentioned in Section 6.1, and a number of other artifacts of this work (including the entire TeX source of this document) can be downloaded from `http://mil.codeslower.com`. The author can also be contacted via e-mail at `jgbailey@gmail.com`.

Colophon

We typeset all elements of this document using TeX and LaTeX. We created our graphical figures with the Ti*k*Z library. We converted our literate Haskell sources to TeX code with Hinze and Löh's lhs2TeX pre-processor. We used Chris Monson's LaTeX `Makefile`[1] to orchestrate the compilation process that produced this PDF.

This document uses 12-pt Palatino for body text and 12-pt Helvetica for headings and titles. Margins, line-spacing and font size conform to guidelines given by Portland State University's Office of Graduate Studies.

We created this version of the thesis on June 5, 2012.

[1] Available from `http://code.google.com/p/latex-makefile/`.