A Programming Language for Precision/Cost Tradeoffs

by

Matthew D. Steele

S.B., Massachusetts Institute of Technology (2008)

Submitted to the Department of Electrical Engineering and Computer Science

in partial fulfillment of the requirements for the Degree of

Master of Engineering in Electrical Engineering and Computer Science

at the Massachusetts Institute of Technology

May 2009

Author .....
Department of Electrical Engineering and Computer Science
May 27, 2009

Certified by .....
Saman Amarasinghe
Associate Professor of Computer Science
Thesis Supervisor

Accepted by .....
Arthur C. Smith
Professor of Electrical Engineering
Chairman, Department Committee on Graduate Theses

A Programming Language for Precision/Cost Tradeoffs

by

Matthew D. Steele

Submitted to the
Department of Electrical Engineering and Computer Science

May 27, 2009

In partial fulfillment of the requirements for the Degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

Many computational systems need to deal with various forms of imprecision and uncertainty in their data; it is also the case that many systems, especially mobile and distributed systems, must be able to trade off the precision of their data and operations against the cost of performing those operations. Unfortunately, for many applications, trying to make these tradeoffs severely complicates the program, because there does not yet exist a programming model that gives the programmer the ability to easily describe the relevant tradeoffs between precision and cost of operations or to express in an algorithm what tradeoffs are appropriate under what circumstances. This paper lays a solid foundation for exploring such programming models by introducing and analyzing a simple core abstraction on which others can be based. We determine what sorts of strategies are and are not possible within this abstraction, and discuss what specific difficulties must be overcome in future work in order to extend the abstraction to encompass a larger class of programs.

Thesis Supervisor: Saman Amarasinghe
Title: Associate Professor of Computer Science

Acknowledgments

I didn't quite know what to expect when I walked into Prof. Saman Amarasinghe's office in the Spring of 2008 to ask for an M.Eng. He accepted, and mentioned that he had an idea for a project for me: a programming language where each value was associated also with a precision and a cost, so that various kinds of tradeoffs between precision and cost could be made. It sounded like a straightforward problem, and one that would be both interesting and useful to solve, so I took it. By six to eight months later, I had discovered that this problem was, in fact, fantastically difficult, and had all but given up hope that any useful result was even possible. It was because of Prof. Amarasinghe's continued faith in me and in this project, and his relentless assistance in paring down the scope of my research to the much more managable core that this thesis represents (despite my continued attempts to make things unduly complicated), that this thesis exists. Thank you, Saman, for not accepting any less from me.

This research would never have gone anywhere without the technical contributions of the other members of the Cbass project and of several other members of the lab. Saman Amarasinghe, of course, led the project, gave my research focus, and continually guided me towards a feasible solution. Jean Yang, one of the original members of the Cbass project, frequently went above and beyond her own work to give me suggestions for what directions to take and what potential solutions to try. Matt Aasted, another member of the Cbass project, did a great deal to find ways for my part of the project to be of practical use, and to relate my abstract model to real-world robotics, as did Arjuna Balasuriya, who provided the original motivation for the project. Armando Solar-Lezama, despite not being part of the project, generously took several hours of his time to discuss my research with me, and his insights into the problem helped lead to several of the key ideas for incremental and partial resolution. Martin Rinard helped to bring me down to Earth early on, and induced me to try to formalize and solidify the vague and primitive early ideas that, without him, I might not have gotten past.

Of course, it would have been impossible for me to spend a year working on this project without the generous financial support of Saman Amarasinghe and Arjuna Balasuriya, nor without the MIT EECS department, which graciously accepted me into the M.Eng. program and allowed me to spend one more year at the Institvte.

I give my profuse thanks to my reviewers, who caught an enormous number of errors that had gotten by me and gave many excellent suggestions for other improvements: Saman Amarasinghe, Martí Bolívar, Ben Charrow, and Guy Steele. Without them, this thesis would doubtlessly have been quite incomprehensible.

Finally, I would like to extend a special thanks to my friends and family who have supported me over the past year. I would like most of all to thank Stephanie Chin, who has been extraordinarily patient and encouraging throughout this process; I thank my parents, Guy and Barbara Steele, who have never failed to support my educational ambitions and to encourage me the entire way; Matt Aasted, who perhaps more than anyone helped me to retain my sanity during the more discouraging parts of this process; Mary

McDavit, who was always supportive and sympathetic; and all my other friends—Ben Charrow, Martí Bolívar, Aparna Rolfe, Kayla Jacobs, and all the residents of 2nd West—who have been ever empathetic to my occasional sleeplessness. Thank you everyone for putting up with me.

Contents

THIS PAGE INTENTIONALLY LEFT BLANK

List of Figures

THIS PAGE INTENTIONALLY LEFT BLANK

Chapter 1

Introduction

Many computational systems need to deal with various forms of imprecision and uncertainty in their data [4, 9, 14, 19, 22]. It is also the case that many systems, especially mobile and distributed systems, must be able to trade off the precision of their data and operations against the cost of performing those operations [9, 19]. Unfortunately, at least for some applications, such as sensor networks, there does not yet exist a programming model that gives the programmer the ability to fully describe the relevant tradeoffs between precision and cost of operations or to express in an algorithm what tradeoffs are appropriate under what circumstances [19].

1.1 Precision vs. cost

Suppose that we have a robot—we'll call it Robby—that needs at various times to know its location. Robby has a number of different sensors, some of which can be used to infer its (approximate) location, but reading data from a sensor requires additional battery power to be spent. Robby's programmer—we'll call her Alyssa—wrote two different functions to determine Robby's location. The first function uses Robby's GPS receiver; this returns a very accurate result, but requires quite a bit of power, because it requires a satellite connection. The second function uses an ultrasonic sonar ping sensor and compares the results against Robby's internal map; this gives only a rough approximation, but requires little power. Which of the two location-approximation functions should Alyssa use when writing other code that needs to know Robby's location?

The answer, of course, depends on the context in which the approximation function is being used. If an accurate result is needed, then Alyssa should use the more accurate function; if a rough answer will do, then Alyssa should use the cheaper function. There is a tradeoff, here, between precision and cost. In a large enough program, Alyssa will probably need both, in different places. She chould choose which one to use manually in each case, but it would be nice if she could just inform the compiler that both functions are approximations of the same value, and let the compiler choose the most sensible function

in each case.

The problem becomes more complicated when the constraints on the required precision are not local. That is, it may not be clear at the call site of the location approximation function how much precision is needed—to determine that, the compiler might need to look at the greater context in which the call appears. Worse, the amount of precision needed from the location approximation function might depend on *other* choices of approximation functions made elsewhere in the program. Even worse, some choices may need to depend on data that is not available until run-time. For example, consider the program diagrammed in Figure 1-1; in two places, we have a choice between two different subroutines, but the

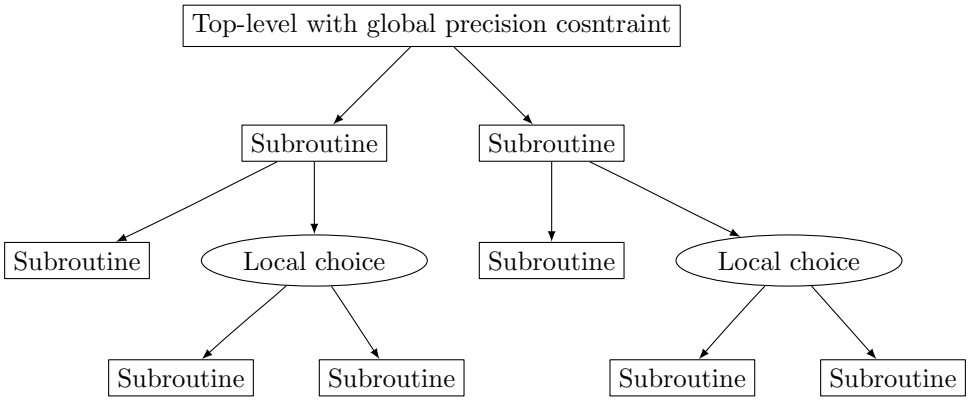

Figure 1-1: The dependency tree of a program with local choices and global precision contraints

constraint on precision appears at the top-level, so we must make these choices together so as to satisfy the global constraint. Globally optimizing all of these choices and splitting the analysis between compile-time and run-time is potentially an extremely difficult problem—all the more reason why Alyssa would want the compiler to do it for her. This work aims at gaining a foothold in this problem.

1.2 Cbass

Cbass (pronounced like "sea bass") is a project that was started in the Commit (**Com**pilers at **MIT**) group at MIT in mid-2008 to provide a programming model for precision/cost tradeoffs, (so named because the initial implementation efforts of the project focused on generating C code to be used in underwater robotics). In order to make it easy for programmers to *express* the relevant tradeoffs in their programs in ways that allow the compiler and/or run-time system to take advantage of those tradeoffs—much as some modern type systems provide abstractions that allow programmers to express invariants in their programs in ways that allow the compiler to take advantage of these invariants for purposes of efficiency [13]—we need to first develop abstractions on which to base such programming models. This work provides a solution to this issue by describing, formalizing, and solving effectively a core subset of the more general problem of finding an effective programming model for precision/cost tradeoffs, and lays

the foundation for future work in this area.

1.3 Contributions of this work

This work has made the following contributions to the field:

- I have described and formalized an abstract model of computation that allows us to view precision/cost tradeoffs in (simple) programs as a straightforward and well-defined optimization problem, in Chapter 2.
- I have designed a simple programming language, *Ccow*, for expressing instances of this computation model, and have implemented an effective compiler for this language; these are described in Chapter 3.
- I have designed, analyzed, and implemented a variety of algorithms for fully resolving "choices" in Ccow programs at compile-time, and have discussed their relative merits, in Chapter 4.
- I have designed, analyzed, and implemented a variety of algorithms for *partially* resolving choices in Ccow programs at compile-time, allowing full resolution to be completed at run-time, and have discussed their relative merits, in Chapter 5.
- I have related this research to other work in the field, and have demonstrated its novelty, in Chapter 6.
- I have drawn conclusions about the usefulness of this and similar models, and have identified future lines of research in this area that seem likely to be fruitful, in Chapter 7.

THIS PAGE INTENTIONALLY LEFT BLANK

Chapter 2

The CC Trees Abstraction

The results of this work are based on an abstracted representation of programs called a *CC tree*; this chapter aims to introduce the reader to this abstraction. The primary goal of this abstraction is to develop a small core problem that we can easily reason about; it is a minimal representation of programs that captures the ideas of *precision and cost* of program values, *local choices* between equivalent approximations, and a *global constraint* on precision. Because of this minimality, as a means of representing the kinds of things we usually think of as programs, CC trees are somewhat limited; however, we will look at some extensions to the abstraction that would allow it to capture more kinds of programs in Section 2.3, and later on, we will discuss why this work has not yet deeply investigated these extensions, in Section 4.4.

2.1 CC Trees

A CC tree is a rooted tree that represents an instance of a particular class of programs; each node of the tree corresponds to a small piece of the program. First, we will introduce the slightly more general abstraction of a *program tree*, and then move on to describe what the "CC" in "CC tree" means.

2.1.1 Program Trees

A *program tree* is a rooted tree. The leaves of the program tree are called *inputs*. Each internal node of the tree indicates how to combine results from its children into a new result, and the root of the tree produces the final result of the program from its children. Figure 2-1 shows the tree for a program that adds two numbers.

When a program tree is placed in some larger context, the inputs may be replaced with some other subtrees, and/or the program tree may replace an input of some larger tree. For example, if we took our addition program from Figure 2-1 and used it as the input to a simple multiplication program, the

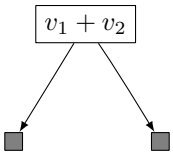

Figure 2-1: A fantastically simple program tree

resulting program tree would look like Figure 2-2. Note that the v_1 and v_2 in each node refer to the respective children of each node, so the two v_1's in Figure 2-2 are referring to two different inputs.

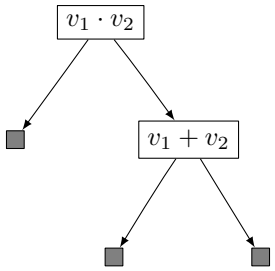

Figure 2-2: An addition program as an input to a multiplication program

Finally, note that just as we can compose program trees to make a larger program tree, we can also do the reverse—any subtree of a program tree can be potentially be viewed as a program that is being used as an input to another program (the rest of the original tree).

Based on these examples, an intuitive view of program trees is that they represent simple *expressions*. For example, the program tree in Figure 2-2 represents the mathematical expression $v_1 \cdot (v_2 + v_3)$, where v_1, v_2, and v_3 are the inputs to the program. Or, to think of it another way, that program tree represents the function "`tree`" in the following C++ code (assuming that the values we are computing are floating-point numbers):

```
double tree(double v1, double v2, double v3) {
    return times(v1, plus(v2, v3));
}
double times(double v1, double v2) { return v1 * v2; }
double plus(double v1, double v2) { return v1 + v2; }
```

The important point made by the above code is that the program tree is representing a *program*. However, the reader may have already noticed that a program tree cannot express shared subexpressions (that would require a program *dag*) or recursion or loops (that would require a potentially infinite program tree, or perhaps a program *digraph*), which places harsh limits on the sorts of programs that can be expressed by a program tree. We will discuss these issues further in Section 2.3.

2.1.2 Precision and cost

So far we have only thought of program trees as manipulating *values*, and producing an output value. In traditional programs, these values are assumed to be exact, and the inputs are provided "for free"—all that the program must do is produce the correct output when given the inputs. However, in this work, we will be examining programs whose input values are *not* exact—and therefore have varying levels of precision—and whose inputs are *not* provided free of charge. Each input will have an associated cost, and the program may need to choose which inputs to use in order to obtain the final result as cheaply as possible. Furthermore, the program will be given a *precision constraint* that it must meet—when making choices, the program must ensure that the precision of the final result value will meet or exceed the constraint.

To this end, each node in the program tree shall have three quantities associated with it: the *value* (or *result*), the *precision*, and the *cost*. Inituitively, the precision gives some indication of how accurate the value is (independent of what the value actually is), and the cost gives some indication of how expensive it was to obtain the value. In general, the value, precision, and cost of each node will be defined in terms of the values, precisions, and costs of its children. In addition to determining the *value* of the root node (which is the final program result), the program must ensure that the *precision* of the root node (which is the precision of the final result) is equal or greater than the precision constraint, all while minimizing the *cost* of the root node (which is the total cost of running the program).

The nodes of the program tree come in several kinds, which we will introduce one at a time.

2.1.3 Input nodes

An *input* node is a leaf of the program tree. On any particular run of the program, an input node has a known precision and cost (which may possibly vary from run to run). It also has a particular value (which almost certainly varies from run to run); however, we cannot know the value of an input unless we pay its cost. Of course, if the result value of a program depends on the value of some particular input, then in order to determine the result we have no choice but to pay the cost of the input. However, *choice* nodes, introduced below in Section 2.1.5, will sometimes give us a choice as to which inputs the program result will actually depend on, which in turn gives us a choice as to for which inputs we must pay.

2.1.4 Combinator nodes

A *combinator* node B consists of two functions, which we will call V_B and P_B. Each function takes n inputs, where n is the number of children of B in the tree. The value v_B of node B is defined to be $V_B(v_1, v_2, \ldots, v_n)$ where v_i is the value of the ith child of B. Similarly, the precision p_B of node B is defined to be $P_B(p_1, p_2, \ldots, p_n)$ where p_i is the precision of the ith child of B (note that this means that the precision of a combinator node does *not* depend on the values of its children; allowing precision functions

that *do* depend on child values is one of the extensions that we will discuss later on in Section 2.3.4). Finally, the cost c_B of node B is defined to be the sum $\sum_{i=1}^{n} c_i$ of the costs of the children of B.

An example of a simple combinator node would be a *subtraction* node S with two children, where the output value is the difference of the input values, and the output precision is the minimum of the input precisions, so that $V_S(v_1, v_2) = v_1 - v_2$ and $P_S(p_1, p_2) = \min(p_1, p_2)$. Other combinator nodes might be considerably more complicated and have large numbers of children.

2.1.5 Choice nodes

The next kind of node, the *choice* node, is perhaps the most important. The value, precision, and cost of a choice node $\mathcal{C}$ is the same value, precision, and cost of just one of its children, to be chosen "automatically" (in practice, for a real computer program, this might mean that the child is chosen by the compiler or run-time system for the program).

To *resolve* a choice node is to choose this child, and to resolve a program tree is to choose the child for each choice node in the tree; the choice (or set of choices) arrived at is called a *resolution*. We say that a particular resolution of a program tree is *admissible* for a given set of inputs (with known precisions) if running the program with those choices on those inputs will yield a final root-node precision that meets the precision requirement. A resolution is called *optimal* for a given set of inputs (with known precisions and costs) if 1) it is admissible, and 2) out of all admissible resolutions, it is the one with the minimum root-node cost. Similarly, a resolution can be called *near-optimal* if it is admissible and has a relatively low (but perhaps not completely optimal) root-node cost.

The goal of this paper is to find efficient algorithms and methods for finding optimal and near-optimal resolutions for program trees.

2.1.6 CC Trees

We could define various kinds of internal nodes other than combinator and choice nodes, and use these in our program trees (in fact, we discuss some other possible node types later on in Section 2.3). However, for most of the rest of this paper, we will concern ourselves only with *combinator/choice (CC) trees*—program trees containing only combinator nodes and choice nodes (and inputs nodes at the leaves). An example of a simple CC tree is shown in Figure 2-3. The choice nodes are marked "$\mathcal{C}$", the combinator nodes are marked with the definitions of their V functions (their P functions are not shown), and the inputs to the tree are marked with gray squares.

Once we know the precisions and costs of the inputs, we can begin to search for an optimal resolution. Once we have chosen a resolution, we can replace each subtree rooted at a choice node with the subtree rooted at the chosen child of the choice node, thus obtaining a program tree containing no choice nodes at all; such a program tree is called *resolved*, and as it has no choices, it can be executed straightforwardly. One simple strategy for finding an optimal resolution is brute force—we can simply try every possible

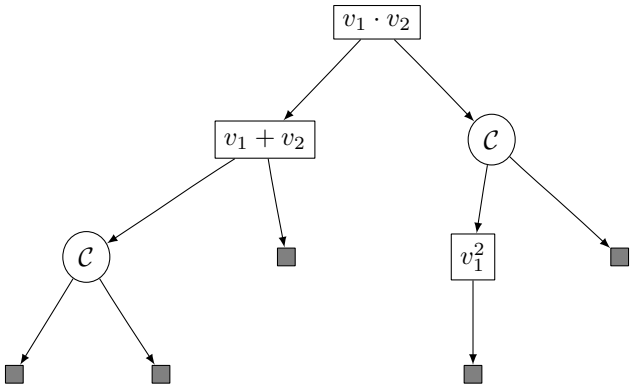

Figure 2-3: A typical CC tree (combinator precision functions and input precisions/costs not shown)

resolution, and for each one, evaluate the root precision. After eliminating all inadmissible resolutions, we pick the remaining one with the lowest cost. Unfortunately, the number of possible resolutions is, in general, exponential in the number of choice nodes in the tree, so this kind of brute force solution may be intractable for program trees with large numbers of choices. Later on, we will explore more clever strategies for resolving CC trees.

2.2 Theoretical difficulty of CC tree resolution

Having introduced the problem of CC tree resolution, we now turn to the question: how computationally difficult a problem is this to solve? To answer this question, we'll need a few more definitions and requirements.

2.2.1 Unicombinator trees

To help us understand the computational complexity of the problem of CC tree resolution, we'll look at a particular class of CC trees that are easy to analyze.

Definition 2.1 (Unicombinator tree)**.** A CC tree T is said to be a *unicombinator tree* if the root node is a combinator node, every child of the root node is a choice node, and every child of each choice node is an input (that is, a leaf).

Figure 2-4 shows an archetypical unicombinator CC tree; the root combinator node R is marked with its precision function P_R. By definition, all unicombinator CC trees have a depth of exactly 2.

The power of unicombinator trees is that the structure of the tree is quite regular, and the resolution behavior of the tree is completely defined by the single root node precision function P_R (as well as by the precisions and costs of the leaves); these features make unicombinator CC trees easy for us to analyze.

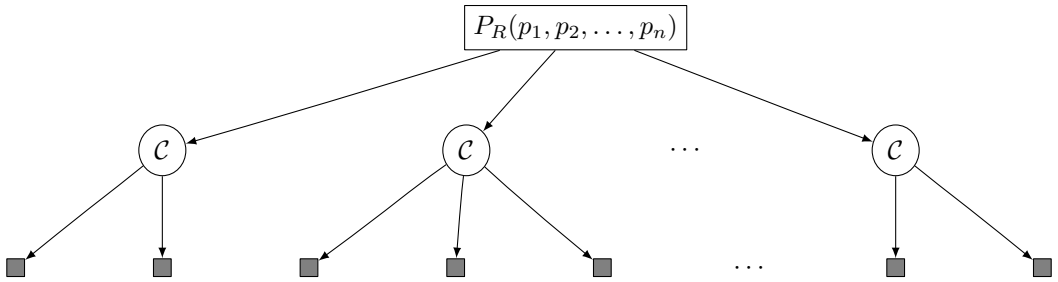

Figure 2-4: A unicombinator CC tree

2.2.2 The monotonicity requirement

Because the resolution behavior of a unicombinator tree T is mostly defined by its root precision function P_R, we would expect that that the diffculty of resolving T would also depend on P_R.

The worst possible case is that P_R is some cryptographically secure hash function. This would imply that the final precision of R is unanalyzable—any change in any input precision could completely change the output precision. This is, of course, a completely contrived case. However, in this pathological situation, the *only* way to find the optimal resolution is the brute force solution of trying every possible combination of choices—finding any better method would be equivalent to "cracking" the cryptographic hash. If there are n choice nodes, each of which has at least two children, then there are $\Omega(2^n)$ possible combinations of choices, so a brute force solution is likely to be intractable for large program trees.

In order to preserve some sanity, we will introduce an additional requirement on all precision functions p that will mitigate this case somewhat:

Requirement 2.2 (Monotonicity)**.** The precision function P of any combinator node must be *monotonically non-decreasing*—that is, increasing any one of its arguments must never cause the result to decrease.

Requirement 2.2 implies that providing a more precise input to a program should never *decrease* the precision of the program output, which seems intuitively obvious. This requirement also eliminates the obvious pathological case described above, since hash functions are generally not monotonic. Henceforth and in future chapters, we will discuss only monotonic precision functions.

2.2.3 Reduction from the knapsack problem

Given the monotonicity requirement, what then is the worst possible case for P_R? As we will soon see, even if P_R is a simple linear function, resolving a unicombinator CC tree can still be very computationally difficult.

Theorem 2.3. *The problem of optimally resolving a unicombinator CC tree with* $P_R(p_1, \ldots, p_n) =$

$\sum_{i=1}^{n} p_i$ *is NP-hard.*

Proof. We will show a reduction to the unicombinator CC tree resolution problem from the *0–1 knapsack problem*, which is known to be NP-hard [10]. Briefly, the 0–1 knapsack problem is this: given a set of items each with a *weight* and a *profit*, and given a knapsack *capacity*, find the subset of items of maximum total profit such that the total weight does not exceed the capacity.

Given an instance S of the knapsack problem with n different items and capacity s, we can reduce S to a unicombinator-CC-tree optimization problem (in polynomial time) as follows:

1. Let $w_{\max}$ be the maximum out of all item weights. For each item i in S, with profit f_i and weight w_i, create a choice node C_i with two leaves: one with zero cost and a precision of $(w_{\max} - w_i)$, and one with a cost of f_i and a precision of $w_{\max}$.

2. Create a root node R with the choice nodes as children, and with precision function $P_R(p_1, \ldots, p_n) = \sum_{i=1}^{n} p_i$.

3. Let the precision constraint on the resulting program tree T be $(n \cdot w_{\max} - s)$.

4. Resolve the choices in T. For each choice node C_i, choosing the zero-cost branch corresponds to putting item i in the knapsack, while choosing the other branch corresponds to not putting item i in the knapsack.

To see that this is a valid reduction, note that choosing a zero-cost branch in choice node C_i saves f_i in cost but also reduces the precision by w_i, as compared to choosing the other branch. Finding the lowest-cost admissible resolution would be equivalent to finding the maximum-sum subset of f_i's such that the sum of the w_i's does not exceed s, which is of course equivalent to the original knapsack problem, as desired. □

Corollary 2.4. *Optimally resolving general binary CC trees is NP-hard.*

Proof. We can perform the same reduction as in Theorem 2.3, replacing the degree-n root combinator node with $(n-1)$ binary combinator nodes, each with precision function $P(p_1, p_2) = p_1 + p_2$, arranged into a binary subtree with the choice nodes at the leaves. The full tree will still have only $O(n)$ nodes, and the previous reduction remains valid. □

Thus, even with very simple precision functions, and limits on the branching factor of the tree, resolving CC trees can be computationally intractible for large trees.

2.3 Extensions to the abstraction

The CC trees abstraction, as introduced in this chapter, is an extremely limited representation of computer programs—it doesn't allow for loops, recursion, shared subexpressions, value-dependent precisions, and

many other useful things—and yet, as we have just seen, even the very simple problem of optimally resolving CC trees is quite difficult to solve efficiently. In this section, we'll briefly discuss some extensions one could make to the CC trees abstraction to make it richer and more expressive. Later on, once we have a better intuition for CC trees, we'll reexamine these extensions and discuss why they are beyond the scope of this work, in Section 4.4.

2.3.1 Program dags

Consider the mathematical expression $(a + b) \cdot (a + c)$, which depends on three values: a, b, and c. The variable a is a *shared subexpression* of this expression, because the expression depends on a in two different places. If a were an *input* with an associated cost, we would expect to pay its cost just once, but be able to use its value twice within the expression. Unfortunately, CC trees do not allow for this. If we tried to represent the above expression as a program tree, we would end up with something like Figure 2-5.

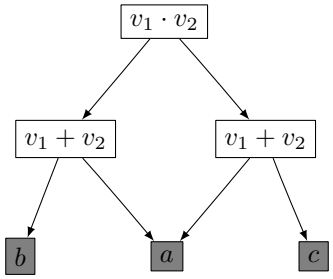

Figure 2-5: A simple program dag

Unfortunately, Figure 2-5 is not a program tree—it is a program *dag*, or directed acyclic graph.[1] We could easily define a CC dag abstraction that was built on program dags, but as we will see later on in this paper, this turns out to make the problem much more difficult. We will discuss the reasons for this in greater depth later on, in Section 4.4.1, but the difficulties stem from the fact that in a CC dag, the *cost* of an individual subtree (or rather, a subdag) is not well defined, as it is dependent on decisions made at choice nodes lying outside this subtree, because we pay only once for each shared input.

2.3.2 Recursion/loops

Consider the factorial function $f(x) = x!$. In a computer program, this function can be written recursively (or with a loop; for simplicity, we will consider just the recursive definition for now). One possible definition is:

$$f(x) = \begin{cases} 1, & x \leq 1 \\ x \cdot f(x-1), & \text{otherwise} \end{cases}$$

[1] As one reviewer pointed out, in this case, we could simply merge the three combinators into a single combinator with three children, yielding a program tree. However, for more complicated program dags containing choice nodes, this isn't always possible.)

If we tried to represent the factorial function as a program dag, we would end up with something like Figure 2-6. The vertical ellipsis in the figure is to suggest that the pattern repeats (potentially) indefinitely, leaving us with an infinite program dag.

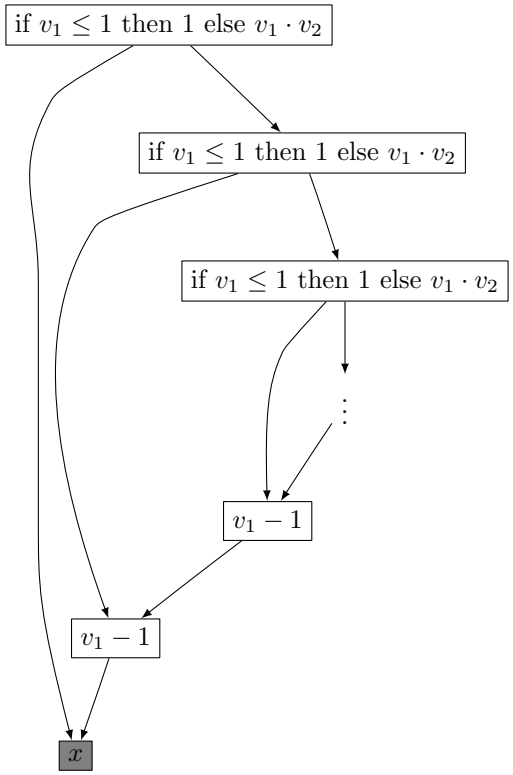

Figure 2-6: An infinite program dag for the factorial function

We could probably come up with a better way to represent this structure, perhaps as a program *digraph* where cycles are permitted (although we would have to be careful when defining the meaning a cycle in the program structure—a naïve interpretation of cycles in a program dag would be that they represent circular data dependencies, which of course we mustn't have). Either way, however, this kind of extension proves difficult to deal with, as we will see in Section 4.4.2.

2.3.3 Conditional nodes

Frequently in real programs, the path of control flow will depend upon run-time values, via *conditionals* (e.g., the `if` keyword or the `?:` trinary operator in C). We could model this in the CC trees abstraction as a combinator node with a value function like this one:

$$V_{\text{cond}}(v_1, v_2, v_3) = \begin{cases} v_2, & v_1 = \mathit{false} \\ v_3, & v_1 = \mathit{true} \end{cases}$$

However, this is not a good model for conditionals, because using this combinator, the cost of the combinator node would include the cost of both branches—that is, we would be paying the cost of inputs in the control branch that we *didn't* take, in addition to paying the cost of inputs that we actually used.

A better approach would be to add a new node type: a *conditional* node. A conditional node functions much like a choice node—it has a number of children, and the precision and cost of the conditional node are the precision and cost of the chosen child. The difference is that the choice of which child to use is compulsory, based on some run-time predicate value. A resolution algorithm would then have to take into account that any branch of the conditional node might be taken. We will discuss the difficulties this would cause, in Section 4.4.4.

2.3.4 Value-dependent precision functions

In the CC trees abstraction, the precision of a node is independent of its value, and the precision of an internal node can always be determined from the precisions of its child nodes, without knowing the *values* of the child nodes, as described in Section 2.1.4. Unfortunately, for some definitions of *precision* that we might want to consider, this is a very harsh restriction.

For example, consider the concept of *variance*. Suppose that we are writing a program that operates on value distributions rather than exact values; we might express the program as a CC tree, where the value of a node in the tree is the mean of an associated distribution, and the precision of a node is the (inverse of) the variance of the associated distribution. When adding two (independent) distributions, we can calculate the new mean by adding the two original means, and we can calculate the new variance from the two old variances, so we could easily define a combinator node in our program for adding two distributions. Similarly, when multiplying two distributions, we can calculate the new mean by multiplying the two original means; however, the new variance will depend both on the two original variances *and* on the two original means. Thus, we *cannot* define a combinator node within the CC trees abstraction for multiplying two distributions, and so any CC-tree-based program that manipulates distributions will be very limited.

It is apparent, then, that we would like to be able to allow precision functions to depend on child values as well as child precisions. We'll discuss why this is not permitted in the CC trees abstraction in Section 4.4.3.

2.4 Summary

Now that we have theoretical basis for CC trees, we can start looking at a practical compiler implementation for dealing with them. We'll take a look at *Ccow*, a simple programming language for expressing and resolving CC trees, in Chapter 3, and then look at how the Ccow compiler resolves CC trees in Chapters 4 and 5.

Chapter 3

The Ccow Language

Ccow (pronounced like "sea cow") is a very simple programming language for specifying and resolving CC trees.[1] A Ccow program consists of a description of one or more CC trees, including information about the structure of each tree, the precision and costs of the inputs, and the precision functions of the combinators. A Ccow program does *not* include information about, for example, the value functions of the combinators—instead, the output from a Ccow compiler is intended to be linked with C++ code describing these missing pieces. The Ccow program includes only that information which is necessary to produce a resolution of each CC tree at compile-time, or else to produce a strategy for producing a resolution at run-time.

3.1 Syntax

Here we give a quick rundown of the Ccow language syntax—not a formal specification, but hopefully enough to allow the reader to easily understand the code snippets in later sections. A full BNF grammar for the Ccow language, as implemented by the Manatee compiler (see Section 3.3), can be found in Appendix A.

In order to make parsing incredibly simple (so that implementation time could be spent elsewhere), and to make the structure of CC trees obvious, Ccow uses a vaguely Lisp-like syntax, where each declaration or expression consists of a sequence of items within parentheses. For example, a typical expression in Ccow might look like this:

```
(comb1 (choice inp1
               (comb2 inp2 inp3)
               inp4)
       (comb3 inp5 inp6))
```

[1] As the reader may have guessed, Ccow was named in analogy with Cbass, and so as to contain the letters "cc" (since Ccow is specific to CC trees).

3.1.1 The basics

Whitespace is ignored in Ccow, except inasmuch as it separates tokens.

A comment is introduced with a semicolon (`;`) and continues until the end of the line.

An identifier consists of letters and digits (no underscores, for reasons discussed in Section 3.2.3), and may not start with a digit. Ccow also contains a handful of reserved words (such as "**`choice`**"); an identifier must not be one of these (the complete list of reserved words is given in Appendix A).

A *precision literal* consists of a decimal number preceded by a tilde (`~`). For example, `~1.62` is a precision literal.

A *cost literal* consists of a decimal number preceded by a dollar sign (`$`). For example, `$31.4` is a cost literal.

3.1.2 Input declarations

An *input declaration* describes an input node of a CC tree and specifies its name, precision, and cost. It does not, of course, specify the value of the input (which presumably changes from run to run), nor even how this value is produced—it is assumed that the output from the Ccow compiler will be linked with definitions of the input value functions later on, as described in Section 3.2.2.

A typical input declaration looks something like one of these:

```
(input foo ~2.3 $10)
(input bar (~0 ~4) ($4.5 $4.8))
```

The first line in the above code declares an input called "`foo`" with a precision of 2.3 and a cost of 10. The second line declares an input called "`bar`" with precision ranging between 0 and 4 and cost ranging between 4.5 and 4.8.

As the example shows, the precision may be either a single number or a pair of numbers, and so may the cost. If a single number is given, then that is the exact precision (or cost) of the input, and the input declaration is said to have *static precision* (or *static cost*). If a pair of numbers is given, they specify an (inclusive) range in which the actual precision/cost (which is then to be determined at run-time) will fall, and the input declaration is said to have *dynamic precision* (or *dynamic cost*).

3.1.3 Combinator declarations

A *combinator declaration* creates a new combinator function and specifies its name, arity (number of arguments),[2] and precision function P. This combinator function can then be used in any number of combinator nodes in any number of program trees. Note that the value function V is *not* included—it is assumed that the output from the Ccow compiler will be linked with definitions of the combinator value functions later on, as described in Section 3.2.1.

[2]The arity must be non-zero, although one could very easily extend the language to allow zero-arity combinators (thunks).

A typical combinator declaration looks something like one of these:

```
(comb baz minimum 3)
(comb quux sum 4)
```

The first line in the above code declares a combinator function called "`baz`" with an arity of 3 and precision function $P_{\min}(p_1, p_2, p_3) = \min_i p_i$. Similarly, the second line declares a combinator function called "`quux`" with an arity of 4 and precision function $P_{\text{sum}}(p_1, \ldots, p_4) = \sum_i p_i$.

For simplicity, Ccow provides only a small selection of precision functions, which are selected by keyword. The available precision functions are:

- **`minimum`** selects the function $P_{\min}(p_1, \ldots, p_n) = \min_i p_i$.
- **`maximum`** selects the function $P_{\max}(p_1, \ldots, p_n) = \max_i p_i$.
- **`sum`** selects the function $P_{\text{sum}}(p_1, \ldots, p_n) = \sum_i p_i$.
- **`harmonic`** selects the function $P_{\text{har}}(p_1, \ldots, p_n) = 1/\left(\sum_i (1/p_i)\right)$.
- **`constant`** z, where z is a nonnegative decimal number, selects the function $P_{\text{const}(z)}(p_1, \ldots, p_n) = z$.

  For example, **`(input foo constant 1.3 2)`** selects the function $P_{\text{const}(1.3)}(p_1, p_2) = 1.3$.
- **`scale`** z, where z is a nonnegative decimal number, selects the function $P_{\text{scale}(z)}(p_1) = z \cdot p_1$ (in this case, the arity must be 1, and thus does not appear in the input declaration).

  For example, **`(input bar scale 2.1)`** selects the function $P_{\text{const}(2.1)}(p_1) = 2.1 \cdot p_1$.
- **`hash`** *name* selects the function indicated by *name*, which is some (presumably difficult to analyze) precision function provided by the particular compiler implementation.[3]

  For example, in the Manatee Ccow compiler, **`(input baz hash geomean 3)`** selects the geometric mean function $P_{\text{geo}}(p_1, p_2, p_3) = \sqrt[3]{p_1 p_2 p_3}$.

3.1.4 Expressions

Ccow includes several kinds of expressions:

- An *input expression* consists of an identifier naming an input (e.g., `foo`). This corresponds to a leaf of the CC tree.
- A *combinator expression* consists of an identifier naming a combinator function, followed by n other expressions where n is the arity of the function, all within parentheses (e.g., `(baz e1 e2 e3)`). This corresponds to a combinator node of the CC tree (see Section 2.1.4).

[3]The word "hash" is used for this to indicate that the function is impenetrable to analysis, much like the hash function example from Section 2.2.2.

- A *choice expression* consists of the keyword "**choice**" followed by two or more expressions, all within parentheses (e.g., `(choice e1 e2)`). This corresponds to a choice node of the CC tree (see Section 2.1.5).

3.1.5 Tree declarations

A *tree declaration* creates a CC tree and specifies its name, precision constraint, "acceptable cost" (described below), and a single expression representing the structure of the tree. The compiler is required to resolve the tree such that the precision constraint is met, or else declare that this is impossible. Furthermore, the cost should be as low as possible. The "acceptable cost" tells the compiler that any cost less than this is fine even if it isn't quite optimal, which leaves room for approximate solutions.

A typical tree declaration looks something like this:

```
(tree foobar ~3.4 $0
  (comb3 (comb2 inp1 inp2)
         (choice (comb1 inp3) inp4)
         (comb2 (choice inp5 inp6)
                (choice inp7 inp8))))
```

Notice that this tree has an "acceptable cost" of zero, which tells the compiler that a completely optimal solution is desired (in this situation, a non-optimal solution would be acceptable only if it had zero cost, in which case it would, in fact, be optimal). Figure 3-1 shows the CC tree described by this snippet of code.

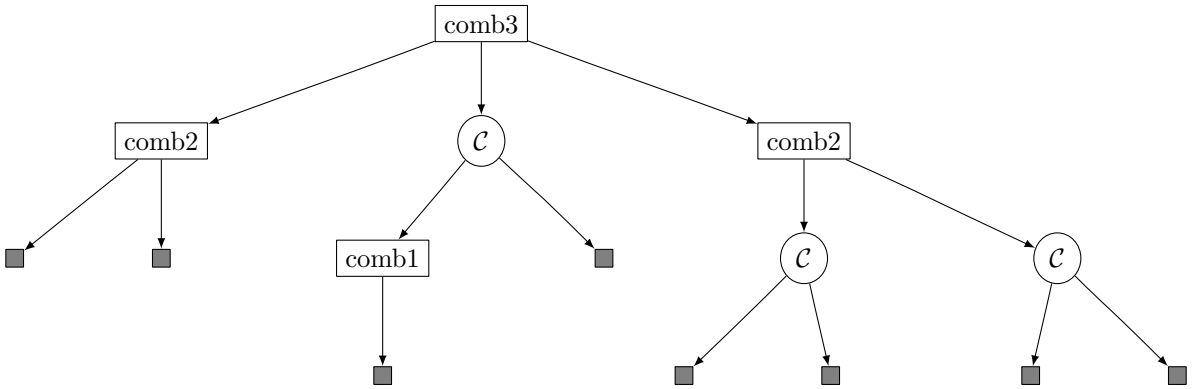

Figure 3-1: A CC tree described by Ccow code

3.2 Semantics

A Ccow program describes the structure, precision functions, and inputs of one or more CC trees; what it does not define is the value functions and the code that produces the value of each input. Thus, the output of a Ccow compiler must be an object file (or a C++ source file, which can then be compiled to

an object file by a C++ compiler) that can be linked with definitions of the combinator value functions and the input functions to form a complete executable. The semantics of the Ccow language are, then, the requirements upon these definitions with which the Ccow output will be linked, and the guarantees on the behavior of the resulting executable. The rest of this section describes these requirements and guarantees.

3.2.1 Value functions

For each combinator declaration, Ccow requires that there be defined externally a function with:

- the same name as the identifier of the declaration,
- a return type of `double`, and
- n arguments, each of type `double`, where n is the arity of the precision function specified by the declaration.

This function will act as the value function for any combinator node using this declaration; when called with the values of the n children of such a node (in order), the function should return the value of the node. It is best to ensure that the function is free of side effects, as Ccow makes no particular guarantees about evaluation order. It is okay for the function to be computationally expensive, but keep in mind that Ccow does not take the computational expense of combinators into account when searching for a good resolution.

3.2.2 Input functions

For each input declaration, Ccow requires that there be defined externally a function with:

- the same name as the identifier of the declaration,
- a return type of `double`, and
- no arguments.

When called, this function should return the runtime value of the corresponding input. It is assumed that *each* call to the function incurs a "cost" equal to the cost of the input—for example, on a robot, invoking an input function may query some physical peripheral device, the use of which consumes substantial battery power. It is *not* assumed that each call to the function returns the same value.

3.2.3 Dynamic precision/cost of inputs

For each `input` declaration, Ccow may require up to two additional functions to be defined externally. If the declaration has dynamic precision, then Ccow requires an externally defined function with:

- a name consisting of the identifier of the declaration concatenated with the string "`_prec`",[4]
- a return type of `double`, and
- no arguments.

When called, this function should return the precision of the corresponding input. The number returned *must* be within the inclusive range specified by the input declaration, or else the behavior of the program is undefined. Furthermore, if called multiple times within the same run of the program, the function must always return the same number. The function is free, of course, to return different numbers on different runs of the program.

Similarly, if the declaration has dynamic cost, then Ccow requires an externally defined function with:

- a name consisting of the identifier of the declaration concatenated with the string "`_cost`",
- a return type of `double`, and
- no arguments.

When called, this function should return the cost associated with the corresponding input. This function is subject to the same restrictions as the previous function—it must return a value within the specified range, and must always return the same value within any particular run of the program.

3.2.4 Tree functions

We now turn to the behavior of a Ccow program. For each tree declaration, the compilation a Ccow program will provide a definition for a function with:

- the same name as the identifier of the declaration,
- a return type of `double`, and
- no arguments.

When called, this function will call a subset of the input functions, and produce a result, corresponding to a resolution R of the CC tree described by the corresponding tree declaration, where R has the following properties:

- R will be *admissible*—that is, it will have a precision greater than or equal to the constraint specified in the tree declaration (unless no such resolution exists, in which case the Ccow program is free to signal an error either at run-time or at compile-time).

[4]Note that because Ccow identifiers are not permitted to contain underscores (as per Section 3.1.1), we avoid the possibility of a name conflict here (we need not worry about having one input named "`foo`" and another named "`foo_prec`").

- R will either be *acceptable*—that is, have cost less than or equal to the "acceptable cost" specified in the tree declaration—or else it will be *optimal*—that is, it will have the lowest cost out of all admissible resolutions. Put another way, if no admissible resolution is acceptable, then the function must use an optimal resolution, but otherwise it may use any (possibly non-optimal) resolution that is both admissible and acceptable.

The tree function will *not* call any input functions other than those depended on by the chosen resolution, and will not call any input function more times than necessary. It will also call the necessary combinator value functions—not more times than necessary—but the evaluation order is not specified. It is free to query the dynamic precision/cost functions of any input that has them.

It is up to the particular Ccow compiler implementation as to how the resolution is determined; in particular, there is no requirement as to whether the resolution is determined statically at compile-time, or dynamically at run-time, or a mixture of both. In fact, the implementation created as part of this work includes different mechanisms (which the user can select between via a command-line flag) for choosing a resolution, in all three of these categories.

3.3 The Manatee Ccow compiler

This work includes an implementation of the Ccow language called *Manatee*, written in Haskell. The Manatee compiler operates in four primary phases:

1. **Parsing.** First, Manatee reads in the file and parses it into an abstract syntax tree (AST). Thanks to Ccow's simplistic syntax, the parser is extremely simple and the AST is straightforward.

2. **Conversion.** Next, Manatee converts the AST into an intermediate representation (IR) that (fairly directly) represents the CC tree(s) described by the program.

3. **Resolution.** This phase attempts to resolve the IR representing each CC tree into another IR representing a program tree without any choice nodes; this phase is the real meat of the compiler. Manatee includes a number of different algorithms for accomplishing this phase, each with different tradeoffs, and allows the user to select which algorithm will be used via a command-line flag.

4. **Code generation.** Finally, Manatee generates C++ code for each resolved tree, which can then be passed into a standard C++ compiler such as GCC. Once the tree is resolved, this phase is fairly trivial.

In order to better mirror the structure of this paper, the compilation path forks at the Conversion stage. For a given input program Manatee uses one of two different IRs—one for programs containing *no* dynamic precisions or costs for inputs, and one for all other programs—and has a completely different set of algorithms used in the Resolution phase for each IR type (the algorithms used in the completely static

case are discussed in Chapter 4, and the algorithms used in the dynamic case are discussed in Chapter 5). In particular, for a program containing dynamic precisions and/or costs, the Resolution phase may not be able to remove all choice nodes at compile-time, in which case it must produce a strategy for completing the resolution at run-time, and the Code Generation phase must then generate the code to execute this strategy (the specifics of this are described in Chapter 5).

Chapter 4

Fully Static Resolution

Static resolution refers to resolving a CC tree "at compile time." In a practical sense, this means that we take a CC tree, resolve it (statically) once, and assume that we can then run the resulting program many times, and the resolution we chose will be optimal (or at least, as good as it ever was), on every run. Throughout this chapter, we will assume that we know the exact precision and cost of each input—that is, that every input has static precision and cost, as defined in Section 3.1.2. We'll tackle the problem of handling programs that must be resolved differently on each run—that is, programs with inputs that have dynamic precision and/or cost—in the next chapter.

Ultimately, the job of the Manatee Ccow compiler is to resolve CC trees; however, there are many different algorithms to accomplish this, with various tradeoffs, as we will see in this chapter. Manatee allows the user to select which algorithm, or *resolution function*, to apply, using a command-line flag. The available resolution functions are described in detail below.

4.1 Brute force resolution

We know from Corollary 2.4 that optimally resolving even relatively simple CC trees is NP-hard in general. It would seem, then, that if we want a completely optimal resolution, at best we can hope for only a "brute-force" solution that simply considers all possibilities and picks the best one. Even so, there are some fairly interesting techniques we can use to improve on the most obvious approach, as well as tricks we can use if we do not require a completely optimal solution, as we will see in this section.

4.1.1 Straightforward brute force

First, let us consider the most obvious, straw-man resolution algorithm. This algorithm can be simply described in steps as follows:

1. Generate a list of *every* possible resolution of the given program tree, each annotated with its final

precision and cost.

2. Remove non-admissible resolutions (that is, any resolution whose precision is less than the given constraint) from the list.

3. If any resolutions remain, return the one with the lowest cost; otherwise, declare that no admissible resolution exists.

It is quite obvious that this algorithm will yield the lowest-cost admissible (that is, optimal) resolution of the given tree for the given constraint. It is also obvious that the runtime of this algorithm is $\Omega(r)$, where r is the total number of possible resolutions. Unfortunately, r is, in general, exponential in the number of choice nodes in the tree, which suggests that this algorithm will be very slow for trees with many choices.

The Manatee Ccow compiler includes an implementation of this (naïve) algorithm as the `brute` resolution function. The only nontrivial part of this algorithm is step 1: generating all possible resolutions. Manatee does this with a recursive function called `allResolutions`. When called on a tree T with root node R, `allResolutions` operates as follows:

- If R is a *leaf* node (in which case R is the only node in T), then return a singleton list of the trivial resolution of T.

- If R is a *choice* node, then recursively call `allResolutions` on each child subtree $S_1, S_2, \ldots, S_k$ of R to get lists $L_1, L_2, \ldots, L_k$ of resolutions. Return the union (that is, the concatenation) $\bigcup_{i=1}^{k} L_i$ of the lists—because R is a choice node, any one of these resolutions is a valid resolution of T.

- If R is a *combinator* node, then recursively call `allResolutions` on each child subtree $S_1, S_2, \ldots, S_k$ of R to get lists $L_1, L_2, \ldots, L_k$ of resolutions. Return the cartesian product $\bigotimes_{i=1}^{k} L_i$ of the lists—because R is a combinator node, the valid resolutions of T are every possible combination of one resolution from each list L_i.

The runtime of the `allResolutions` function is slightly tricky to analyze; however, we already know that number of resolutions produced, and therefore the runtime, must be at least exponential in the number of choice nodes in the whole tree. In other words, it can be quite slow (and indeed, our benchmarks confirm this); so, next we'll look at a technique that can mitigate some of the exponential blow-up.

4.1.2 Eclipse pruning

In the previous algorithm, we had to consider *every* possible resolution of the tree—here, we'll see a technique that lets us ignore "useless" resolutions.

Definition 4.1 (Eclipsing)**.** Suppose we have two resolutions R_1 and R_2 of a CC tree T, with final precisions p_1 and p_2 and costs c_1 and c_2. Now suppose that $p_1 \geq p_2$ and $c_1 \leq c_2$; that is, R_1 is at least as precise and at least as cheap as R_2. In this case, we say that R_1 *eclipses* R_2.

If R_1 eclipses R_2, then obviously, R_2 is not worth further consideration, because it is at least as bad as R_1 in every way (note that, as a quirk of the definition, two resolutions with the same precision and cost each (trivially) eclipse the other, which just means that either one can be discarded since the other is just as good). The key to using this seemingly trivial fact to our advantage lies in Requirement 2.2, the monotonicity requirement.

Theorem 4.2. *Suppose we have a node N that appears in CC tree T, and let $\mathcal{R}$ be the set of resolutions of the subtree rooted at N. Let $\mathcal{S} \subset \mathcal{R}$ be any (strict) subset of $\mathcal{R}$ such that, for every resolution R_2 in $\mathcal{S}$, there exists some resolution R_1 in $\mathcal{R}' = \mathcal{R} \setminus \mathcal{S}$ such that R_1 eclipses R_2. Assuming that T has at least one admissible resolution, then there exists an optimal resolution of T in which no member of $\mathcal{S}$ appears.*[1]

Proof. The theorem can be proved by induction on the depth of node N. As a base case, if N is the root of T ($\text{depth}(N) = 0$), then the theorem follows immediately. Each optimal resolution of T is either in $\mathcal{R}'$ or in $\mathcal{S}$; there is at least one optimal resolution, so either it is in $\mathcal{R}'$, or it is in $\mathcal{S}$, in which case it is (trivially) eclipsed by some other (optimal) resolution in $\mathcal{R}'$.

For the inductive step, assume that the theorem holds for all depths less than $\text{depth}(N) = k$, and let node M be the parent of N. For each resolution R_2 in $\mathcal{S}$, let $R_1 \in \mathcal{R}'$ be a resolution that eclipses it. Now, for each resolution Q_2 of the subtree rooted at M that contains R_2, let Q_1 be another resolution that is the same as Q_2 except that R_2 is replaced by R_1. Because R_1 eclipses R_2, we have $\text{cost}(R_1) \leq \text{cost}(R_2)$, and therefore that $\text{cost}(Q_1) \leq \text{cost}(Q_2)$; furthermore, we have $\text{prec}(R_1) \geq \text{prec}(R_2)$, and therefore, by the monotonicity requirement, that $\text{prec}(Q_1) \geq \text{prec}(Q_2)$. Thus, Q_1 eclipses Q_2. Let $\mathcal{Q}$ be the set of all these Q_2's—that is, $\mathcal{Q}$ is the set of every resolution of the subtree rooted at M that contains any member R_2 of $\mathcal{S}$. Every one of these resolutions is eclipsed by some other resolution Q_1 not in $\mathcal{Q}$ (because Q_1 does not contain a member of $\mathcal{S}$). Thus, because $\text{depth}(M) < k$, by the inductive hypothesis, we can find an optimal resolution of T that does not contain any member of $\mathcal{Q}$, and that therefore does not contain any member of $\mathcal{S}$. □

Any resolution of a tree T is composed of resolutions of subtrees of T. If each child subtree of some node N in T has a number of possible resolutions, we don't know *a priori* which ones will combine to form the resolution of N that will appear in the optimal resolution of T. In the previous brute-force strategy, we would simply try every possible combination to get the list of resolutions for N. What Theorem 4.2

[1] Unfortunately, the statement of this proof is complicated by the fact that a tree can have multiple optimal resolutions, each of which (trivially) eclipses the others—thus, although there may be an optimal resolution that contains the eclipsed subresolutions, there is always another one that doesn't.

tells us, however, is that if some resolution R of a child subtree of N is eclipsed, then we need not consider resolutions of N that contain R. This reduces the number of resolutions of N that we must consider, which in turn reduces the number of full resolutions of T that we will consider. This technique is called *eclipse pruning*, and Theorem 4.2 allows us to apply eclipse pruning at *every level* of the tree.

The Manatee Ccow compiler includes an implementation of a brute-force resolver using eclipse pruning as the `epbrute` resolution function. This resolver works the same as the `brute` resolver, except that instead of gathering *all* possible resolutions of the tree with the `allResolutions` function, it gathers only the *non-eclipsed* resolutions of the tree, using a recursive function called `usefulResolutions`. When called on a tree T with root node R, `usefulResolutions` operates as follows:

1. First, call `usefulResolutions` recursively on any child subtrees of R, and combine them in the same way as in `allResolutions` (depending on what kind of node R is) to get a list L of resolutions of T.

2. Next, instead of returning L, sort the resolutions in L in *descending* order by precision, breaking ties by sorting in *ascending* order by cost, to obtain a new list L'.

3. Finally, create a new list L'' as follows: run through list L' in order, and for each item, append it to L'' if its cost is strictly less than the current last element of L'' (or if L'' is still empty). Return L''.

Let's quickly show that this algorithm works as advertised:[2]

Proposition 4.3. *The result list L'' of the* `usefulResolutions` *function will contain exactly the set of resolutions from the input list L that are not eclipsed by any other element of L.*

Proof. Any resolution R that was not added to L'' in the final step was eclipsed by the current last element R' of L''—R was not added because R' had equal or lesser cost, and due to the sorting, that R' must have had equal or greater precision. Thus, all non-eclipsed resolutions are included.

Any resolution R that *was* added to L'' was not eclipsed by any other resolution in L, because if another resolution R' eclipsed R, then due to the sorting R' would already have been added to L'', and by transitivity all following resolutions added to L'' would be strictly cheaper than R', and so R (which is at least as expensive as R') could not have been added. Thus, no eclipsed resolution is included. □

The expected runtime of the `usefulResolutions` function is quite tricky to analyze—the extra sorting step would seem to increase the complexity in comparison to the `allResolutions` function, but on the other hand the filtering mitigates the exponential blow-up over large trees. As we will see in the next section, using `usefulResolutions` instead of `allResolutions` makes the `epbrute` much faster than the `brute` resolver in practice.

[2] An anecdote: my first implementation of the `usefulResolutions` function was incorrect in that it could include some eclipsed resolutions in certain situations (which unfortunately was not caught by my unit tests). I didn't notice this bug until I tried to prove here that my code was correct (and discovered that it wasn't!), at which point I replaced it with the algorithm described above.

4.1.3 Intuition

To help build some intuition for how eclipse pruning works, it is helpful to think of resolutions as being points on a plane, where one axis represents precision and the other represents cost. In this case, each resolution point eclipses all resolution points in the quarter-plane starting at the point and extending towards zero precision and infinite cost.

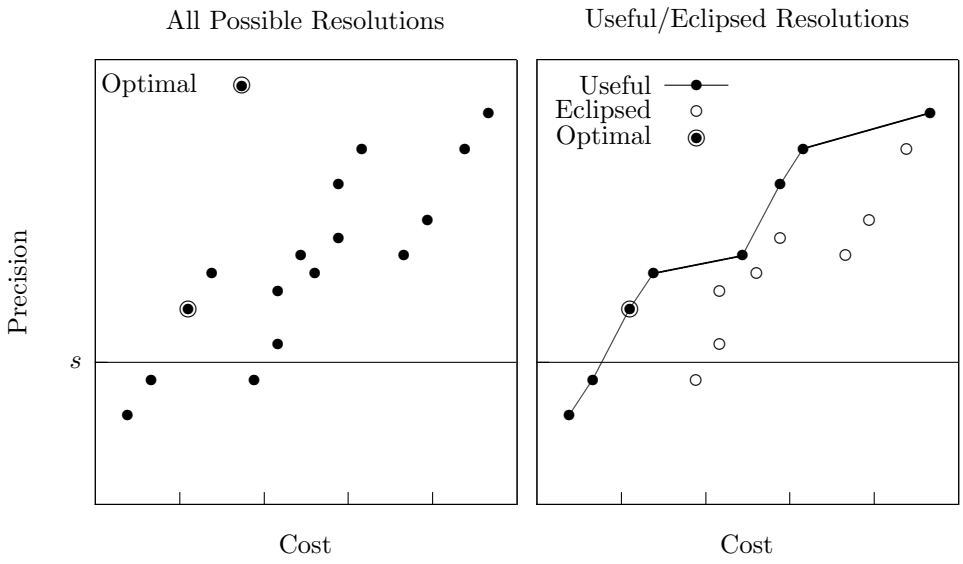

Figure 4-1: Plot of precision/cost of possible resolutions of a CC tree

For example, Figure 4-1 shows, on the left, a plot of the sixteen possible resolutions of some particular CC tree T, and on the right, the same sixteen resolutions with eclipsed resolutions marked with white circles instead of black. Each circle represents a possible resolution, with the horizontal position representing the cost of the resolution, and the vertical position representing the precision. As we can see, the non-eclipsed resolutions form a monotonically strictly increasing curve through precision/cost-space, and all eclipsed resolutions lie below and to the right of this curve. The horizontal line shows the precision contraint s; the optimal resolution is circled on each side. The optimal resolution is (by definition) the resolution of minimum cost that still has precision at least s, which is the leftmost circle that is above the line, as we can see on the left plot; equivalently, the optimal resolution is the *non-eclipsed* resolution of minimum precision that is still at least s, which is the lowest black circle above the line, as we can see on the right plot.

4.1.4 Effectiveness

To test the effectiveness of the `brute` and `epbrute` resolvers (and other resolvers in this chapter), we constructed a number of benchmark Ccow programs, and timed how long it took each resolver to resolve

each benchmark. To do this, we compiled the Manatee Ccow compiler from its Haskell source code with optimizations on, and then ran Manatee with each resolver on each benchmark on a 3.2 GHz Pentium 4 desktop PC. We did not have room to include the full source code for every benchmark in this thesis, but the source code for one of our static-resolution benchmarks is given in Appendix B.

As it turned out, eclipse pruning makes an enormous difference in the running time of the brute force algorithm. We constructed several benchmark CC trees and resolved each one using first the `brute` and `epbrute` resolvers; Figure 4-2 shows the resolution time for each case (on a logarithmic scale).

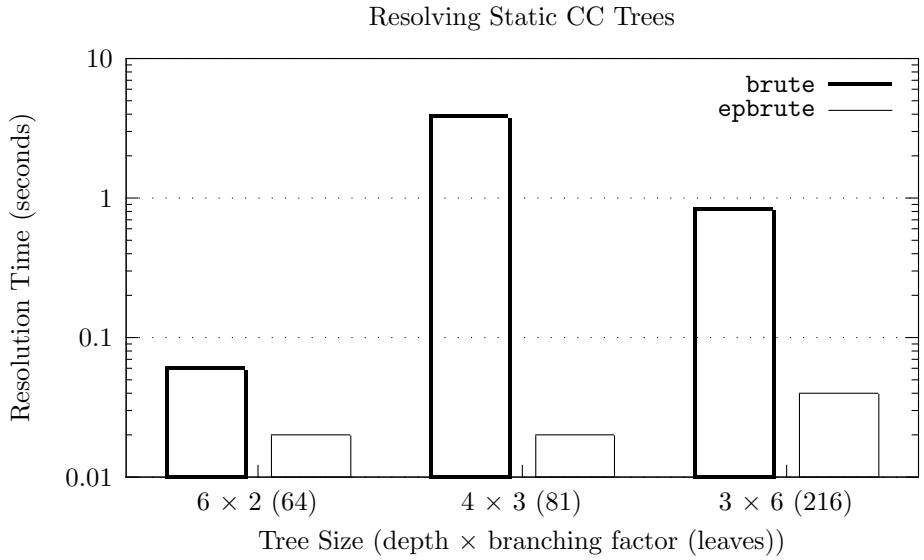

Figure 4-2: Performance comparision of the `brute` and `epbrute` resolvers

As we can see, adding eclipse pruning to the brute force strategy improves the performance by an order of magnitude or two. In fact, even these results are somewhat misleading—the size of the benchmarks shown in Figure 4-2 had to be chosen very carefully in order to be able to compare the two resolvers, because making them even a little larger tended to cause the `brute` resolver to exhaust memory. For example, on a benchmark with depth 4 and branching factor 4 (a paltry 256 leaves), using the `brute` resolver causes the compiler to hang for several minutes before crashing, while the `epbrute` resolver is able to finish successfully in a fraction of a second. Clearly, then, we can conclude that the `brute` resolver is, objectively, not particularly effective,[3] while the `epbrute` resolver is far better, and probably quite usable. Later in this chapter, we will compare the `epbrute` resolver against other resolution algorithms.

[3]At least, the `brute` resolver is not effective as currently implemented. Doubtlessly, one could engineer a more clever implementation with reduced memory requirements, but the point here is that if we use eclipse pruning, this is not even necessary.

4.2 Downward propagation

Although we know that CC tree resolution is NP-hard in general, we might ask whether restricting the problem might make it easier. In particular, the reduction used in Theorem 2.3 made use of the P_{sum} precision function—what if we tried restricting precision functions to an even smaller class of functions than that allowed by Requirement 2.2, and excluded functions such as P_{sum}? In this section, we'll define a class of precision functions that allows us to achieve polynomial-time resolution, and then implement and measure a resolution algorithm that operates on trees restricted to such functions.

4.2.1 Optimal substructure

Part of what makes CC tree resolution difficult is that the problem must be globally optimized with respect to a global constraint. We would be better off if the problem could be made to exhibit *optimal substructure*, so that we could optimize the whole problem by optimizing individual subproblems locally and independently.

For example, suppose we have a CC tree T whose root node R is a combinator node having precision function $P_{\min}(p_1, \ldots, p_n) = \min_i p_i$. In order for R to meet a precision constraint s, all of the children of R must meet the same constraint s. Furthermore, the optimal resolution for the whole tree T consists of the conjunction of the optimal resolutions for each of child subtrees of R. This means that we can optimally resolve T with constraint s by optimally resolving each child subtree of R with the same constraint s and then joining the resolutions together. In this case, we say that we have *propagated the constraint downward* to the child subtrees.

Even if the child subtrees must each be solved by brute force, this scheme can save us an enormous amount of work. For example, suppose that R has n subtrees, each with k possible resolutions (after eclipse pruning), and that obtaining each set of k resolutions requires M time. There would then be k^n possible resolutions of the whole tree T, so resolving T by brute force would require would require $O(k^n + nM)$ time. By contrast, resolving T by optimal substructure would require only $O(kn + nM)$ time.

4.2.2 Formal results

First, we will define formally under what conditions we say we are able to propagate a constraint downward.

Definition 4.4 (Downward-propagatable function)**.** We say that a precision function P is *downward-propagatable* if there exists an algorithm that, given some precision contraint s and a CC tree T whose root node R has precision function P and k children, can resolve T optimally using only: 1) at most $\ell = O(1)$ optimal resolutions for each child subtree S_i of R using some $f_{i,j}(s)$ as the constraint for each

resolution (where $1 \leq i \leq k$ and $1 \leq j \leq \ell$), and 2) at most Poly(k) additional work.

Definition 4.5 (Downward-propagatable tree)**.** We say that a CC tree T is *downward-propagatable* if every combinator node in T has a downward-propagatable precision function.

With these definitions in hand, we can make a statement about the ease of resolving trees with downward-propagatable functions.

Theorem 4.6. *Any downward-propagatable CC tree T with n nodes can be optimally resolved in* Poly(n) *time. If T has a maximum branching factor of $O(1)$, then T can be optimally resolved in $O(n)$ time.*

Proof. We can prove the theorem by structural induction on T. As a base case, note that a leaf is trivially resolved in constant time. For the inductive step, consider a tree T with n total nodes, and with a root node R that has k children. By the inductive hypothesis, we can optimally resolve each child subtree of R, ℓ times (with ℓ different constraints), in Poly(n)ℓ = Poly(n) time. Note that, by necessity, $k < n$; thus, we can resolve all of the child subtrees in a total of Poly(n)k = Poly(n) time, and then optimally resolve R in an additional Poly(k) = Poly(n) time. Thus, the total time to resolve T optimally is Poly(n).

In the case where $k = O(1)$, the inductive step changes slightly. Because T has n total nodes, by the inductive hypothesis we can resolve all k child subtrees once in $O(n)$ time, and all of them ℓ times in $O(\ell n) = O(n)$ time. Finally, after applying an additional Poly(k) = $O(1)$ work, we can obtain the optimal resolution for T. Thus, the total time to resolve T optimally is $O(n)$. □

Theorem 4.6 is certainly good news, because it means that some CC trees can be resolved very efficiently. However, we should note that (in all likelihood) not all precision functions that we are interested in are downward-propagatable.

Corollary 4.7. *Assuming that* P $\neq$ NP*, the precision function $P_{\text{sum}}(p_1, \ldots, p_n) = \sum_{i=1}^{n} p_i$ is not downward-propagatable.*

Proof. This follows immediately from Theorems 2.3 and 4.6. If P_{sum} were downward-propagatable, then we could optimally resolve a unicombinator CC tree with $P_R = P_{\text{sum}}$ in polynomial time; however, that problem is NP-hard, so this would imply that P = NP. □

Thus, if P $\neq$ NP, then P_{sum} is certainly not downward-propagatable; if P = NP, then it probably doesn't matter whether or not P_{sum} is downward-propagatable, since there are probably then better ways to solve the problem.

Finally, we should note that Theorem 4.6 applies only to trees that contain *only* downward-propagatable functions. If a tree contains a non-downward-propagatable combinator node, then we will have to resort to brute force (or some other method) for this node *and all nodes below it*, even if those other nodes are themselves downward-propagatable. The reason for this is that downward-propagation works only

when we have a local constraint, and even one non-downward-propagatable function ruins our ability to propagate the global constraint past it to make it local.

4.2.3 Downward-propagatable functions

Here we describe some precision functions that allow us to use downward propagation. Let R be the root node of some CC tree T, let P_R denote the precision function of R, and let $S_1, \ldots, S_k$ be the child subtrees of R.

- Suppose that $P_R = P_{\text{const}(z)}$, where $P_{\text{const}(z)}(p_1, \ldots, p_k) = z$. Because the precision of R does not depend at all on the precisions of its children, clearly the optimal resolution of T is the cheapest possible resolution, which is formed by taking the cheapest possible resolution of each S_i. This can be achieved by optimally resolving each S_i for a constraint of zero—since any resolution will meet this constraint, the optimal resolution will be the cheapest possible resolution. Clearly, then, $P_{\text{const}(z)}$ is downward-propagatable.

- Suppose that $k = 1$ and $P_R = P_{\text{scale}(z)}$, where $P_{\text{scale}(z)}(p_1) = z \cdot p_1$ and $0 < z < \infty$. Then we can optimally resolve T for constraint s by optimally resolving S_1 for constraint s/z. In fact, more generally, if P_R is any invertible function of one argument, then we can optimally resolve T for constraint s by optimally resolving S_1 for constraint $P_R^{-1}(s)$. Clearly, then, P_R is downward-propagatable.

- Suppose that $P_R = P_{\min}$, where $P_{\min}(p_1, \ldots, p_k) = \min_i p_i$. We already saw this example in Section 4.2.1; we can optimally resolve T for contraint s by optimally resolving each S_i for contraint s.

- Suppose that $P_R = P_{\max}$, where $P_{\max}(p_1, \ldots, p_k) = \max_i p_i$. This one is quite a bit trickier. For R to meet the precision constraint s, only one child needs to meet that constraint locally—all the others can be resolved as cheaply as possible. Our strategy is as follows. First, we resolve each S_i once for constraint s, and once for constraint zero—a constant number of optimal subresolutions for each child. Next, we examine each of the k resolutions formed by combining the constraint-s subresolution for one child subtree with the constraint-zero subresolutions of the other child subtrees, and choose the cheapest one—an amount of extra work that is polynomial in k. It should be clear that this will result in an optimal resolution of T, and thus that $P_{\max}$ is downward-propagatable.

Having seen a handful of examples of downward-propagatable functions, we now turn to the implementation of a resolver that takes advantage of them.

4.2.4 Implementation

The `downprop` resolver is driven by a single, large, recursive function called `resolveDownProp` which takes a precision constraint, an acceptable cost, and an unresolved tree, and produces an optimally resolved tree, or reports that no feasible resolution exists. The reason for the function's size is that it contains special-case code for every downward-propagatable precision function supported by the Ccow language. When called, the `resolveDownProp` function examines the root node R of the given tree T, and behaves as follows:

- If R is a leaf (that is, T contains just one node), then return the trivial resolution of T.
- If R is a choice node, then call `resolveDownProp` recursively on each child with the same precision constraint and acceptable cost; among the children for which a feasible subresolution exists, select the cheapest one (if no child has a feasible subresolution, then T does not have a feasible resolution).
- If R is a combinator node, then examine the precision function P:
  - If $P = P_{\text{const}(z)}$, then find the cheapest (and possibly infeasible) subresolution of each child,[4] and join the resulting subresolutions together.
  - If $P = P_{\text{scale}(z)}$, then there is exactly one child; call `resolveDownProp` recursively on this child with the same acceptable cost, and a precision constraint of s/z, where s is the original precision constraint.
  - If $P = P_{\text{min}}$, then call `resolveDownProp` recursively on each child with the same precision constraint and an acceptable cost of zero, and join the resulting subresolutions together.
  - If $P = P_{\text{max}}$, then call `resolveDownProp` recursively on each child with the same precision constraint and an acceptable cost of zero, and also find the cheapest (and possibly infeasible) subresolution of each child. For each child, consider the resolution formed by joining that child's optimal subresolution with the cheapest subresolutions of the other children. Select the cheapest of these composite resolutions.
  - If P is any other function, use the `epbrute` resolver to resolve T.[5]

  Note that if R is a combinator node, then if any child subtree of R cannot be feasibly resolved, then neither can T be feasibly resolved.

One nice property of the `downprop` resolver is that it is strictly better than the `epbrute` resolver—in some cases, it is more efficient, and at worst, it simply falls back to the `epbrute` resolver. Another feature of note is that the `downprop` resolver, in contrast to all the others, is recursive—although the `epbrute` resolver, for example, uses a recursive helper function to *gather* possible resolutions before choosing the

[4] One simple way to do this is to optimize each child with a precision constraint of zero and an acceptable cost of zero.

[5] In the actual implementation, there's one more special case: if $P = P_{\text{har}}$, then we use the `semiprop` resolver instead of the `epbrute` resolver. However, P_{har} and `semiprop` aren't introduced until Section 4.3.

optimal resolution, the `downprop` resolver is *itself* recursive, which is possible only because it is able to resolve subtrees locally.

4.2.5 Effectiveness

To evaluate the effectiveness of the `downprop` resolver, we had it compete with the `epbrute` resolver on a series of benchmarks of downward-propagatable trees. First, we used trees made entirely of choice nodes and combinator nodes with precision function $P_{\min}$, for which we expected the `downprop` resolver to have a huge advantage; Figure 4-3 shows the resolution time for each case (on a logarithmic scale).

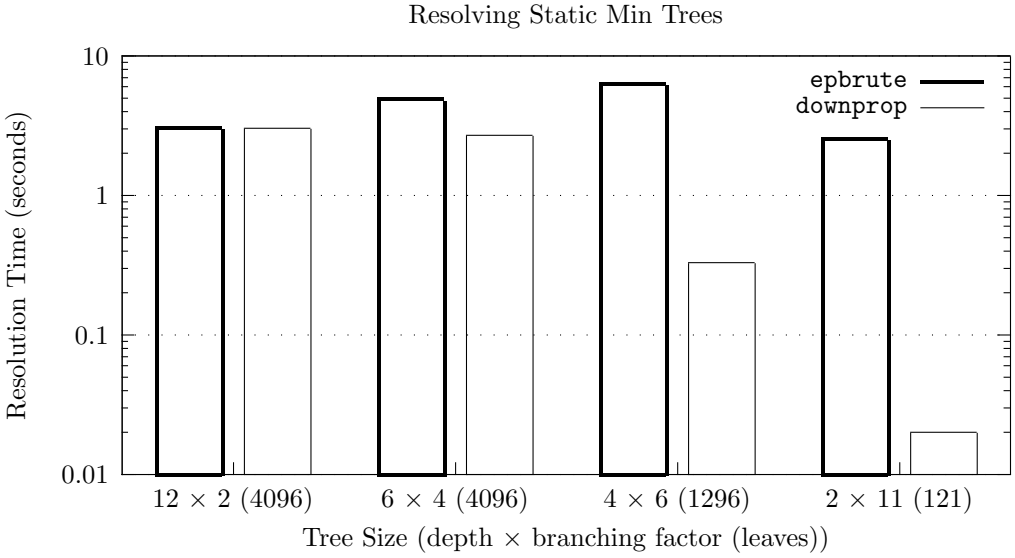

Figure 4-3: Performance comparision of the `epbrute` and `downprop` resolvers on min trees

As we can see, on trees with large branching factors, the `downprop` resolver does considerably better than `epbrute`; however, for smaller branching factors, it does only a little better, even if the tree is large. We suspect this is due to the overhead associated with the `downprop` resolver—the `epbrute` resolver, being so simple, has low constant factors in its running time. These results also indicate that the `epbrute` resolver tends to do better with large depths and small branching factors than with smaller depths and larger branching factors.

Next, we tried using a set of benchmarks of the same size, but with combinator nodes having precision function $P_{\max}$ instead of $P_{\min}$. We expected `epbrute` to do about the same, since it doesn't much rely on which precision functions are being used, and the `downprop` resolver to do somewhat worse, because the $P_{\max}$ case in the `resolveDownProp` function is considerably more complicated than the $P_{\min}$ case. Figure 4-4 shows the resolution time for each case.

What we found was quite a surprise. The `downprop` resolver did only slightly worse on the $P_{\max}$ benchmarks than on the $P_{\min}$ benchmarks; not really enough to notice. Much more surprising was

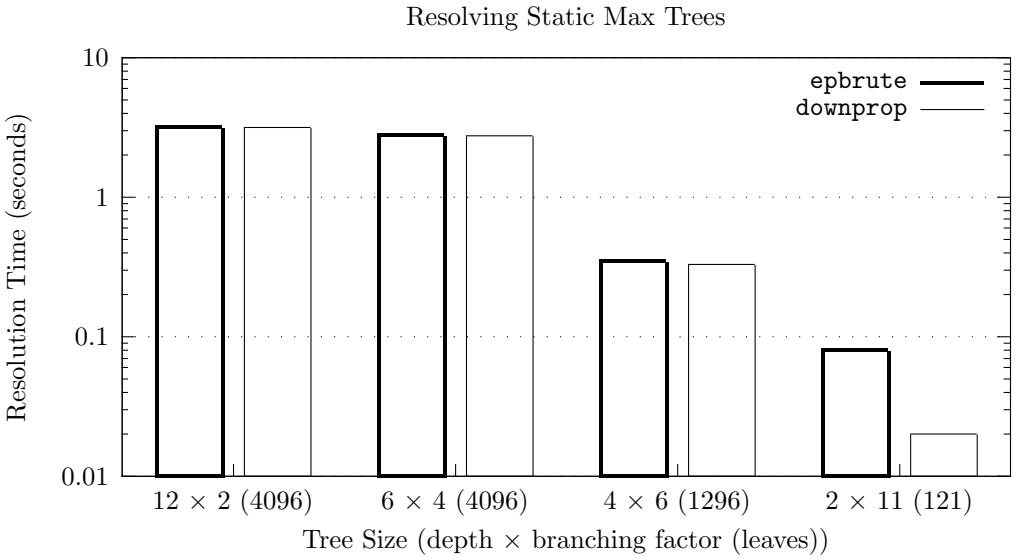

Figure 4-4: Performance comparision of the `epbrute` and `downprop` resolvers on max trees

that that the `epbrute` resolver did far better on the $P_{\max}$ benchmarks than on the $P_{\min}$ benchmarks, nearly matching `downprop` in performance except for trees with very large branching factors. Yet another surprise was that, unlike in the $P_{\min}$ benchmarks, here the `epbrute` resolver tends to do better with small depths and large branching factors than with larger depths and smaller branching factors. We strongly suspect that this is because the $P_{\max}$ function tends to lead to sets of resolutions that are more amenable to eclipse pruning than some other functions.

Our conclusion, after these tests, is that the `epbrute` resolver is actually much more effective than one might think, but that the `downprop` resolver is nonetheless a significant improvement, at least on certain classes of CC trees.

4.3 Downward semi-propagation

Downward propagation is a powerful technique, but it is quite limited in the precision functions to which it can be applied. In this section we introduce a technique that is weaker, but which applies to a wider class of functions.

4.3.1 Harmonic sums

We'll motivate this technique with a new precision function.

Definition 4.8 (Harmonic sum)**.** If $x_1, x_2, \ldots, x_n$ are real numbers with $0 \leq x_i \leq \infty$ for every $1 \leq i \leq n$,

we will refer to the expression

$$P_{\text{har}}(x_1, x_2, \ldots, x_n) = \frac{1}{\frac{1}{x_1} + \frac{1}{x_2} + \cdots + \frac{1}{x_n}}$$

as the *harmonic sum*[6] of $x_1, x_2, \ldots, x_n$.

P_{har} is a particularly useful precision function, because it helps us to capture the notion of *variance*. As we mentioned back in Section 2.3.4, we can represent value distributions within a CC tree by using the value of a node for the mean of a distribution, and the precision of a node for the variance of a distribution. When we want to add two (independent) distributions, we must add the two means to get the new mean, and add the two variances to get the new variance. There is one sticking point with variance, however. Our model specifies that a higher precision indicates a more accurate result—for example, a precision of zero indicates no confidence in the value, and a precision of infinity indicates perfect confidence in the value. However, variance works the opposite way—a variance of zero indicates an exact value, and an infinite variance would indicate a completely unknown value. Thus, we cannot store the variance in the precision number. The obvious solution is to store the *reciprocal* of the variance in the precision number. If we assume that the reciprocal of zero is infinity, and vice versa (which, conveniently enough, is true in IEEE floating-point arithmetic [3]), then this makes the interpretations of zero and infinity work out, and also ensures that less-than/greater-than comparisons between two precisions will be interpreted correctly. Under this representation, to get the precision corresponding to a sum of several (independent) variances, we must take the reciprocal of the sum of the reciprocals of the precisions—that is, the harmonic sum of the precisions!

Unfortunately, P_{har} does not appear to be downward-propagatable. We can easily create a reduction from the knapsack problem, very similar to that of Theorem 2.3, that shows that resolving a unicombinator tree with $P_R = P_{\text{har}}$ is NP-hard,[7] and therefore that P_{har} cannot be downward-propagatable if P $\neq$ NP, by Corollary 4.7.

Fortunately, P_{har} has the following advantage over other non-downward-propagatable functions such as P_{sum}.

Proposition 4.9. *If $P_{\text{har}}(x_1, x_2, \ldots, x_n) \geq s$, then $x_i \geq s$ for every $0 \leq i \leq n$.*

We state Proposition 4.9 without proof. It follows from basic algebra, and is a well-known fact in, for example, electrical engineering, where P_{har} is the function used to calculate the resistance of a group of resistors wired in parallel.

[6] This terminology is apparently slightly non-standard (which is why we have included a definition here), but note that *harmonic sum* is being defined here in analogy to the standard concept of a *harmonic mean*, so this definition should seem reasonable.

[7] To modify the previous reduction to use P_{har} instead of P_{sum}, simply set the precision contraint to $1/s$, give the ith zero-cost branch precision $1/w_i$, and give the ith non-zero-cost branch infinite precision. Showing that this yields a valid reduction is left as an exercise for the reader.

Suppose, now, that we have a CC tree T whose root combinator node R has precision function P_{har}, and we wish to resolve T under precision constraint s. A resolution of T cannot be feasible with respect to the constraint s unless every child subtree of R is also resolved to be feasible with respect to the same constraint s, by Proposition 4.9. Actually, we will likely need the child subtrees to be resolved to even stricter constraints—for example, the only way to have $P_{\text{har}}(p_1, c) \geq s$ is if p_1 is infinite (that is, perfect precision). However, at the very least, this gives us a lower bound on the precision of useful resolutions of the child subtrees of R. We can apply this lower bound (in addition to eclipse pruning) to reduce the space of a brute-force search. In this case, we say that we have *semi-propagated the constraint downward* to the child subtrees—we can't use the constraint to *optimize* each child subtree independently, as we can with "full" downward propagation, but we *can* use the constraint to *prune* each child subtree independently.

4.3.2 Implementation

The Manatee Ccow compiler includes an implementation of a resolution function that uses downward semi-propagation (but *doesn't* use full downward propagation) as the `semiprop` resolver. This resolver is driven entirely by the `semiPropResolutions` function, which is similar in purpose to the `usefulResolutions` function used by the `epbrute` resolver, but is even more discriminating. The `semiPropResolutions` function takes two parameters, a CC tree and a precision bound, and returns a list of all non-eclipsed resolutions whose precisions are greater than or equal to the bound. The `semiprop` resolver simply calls this function with the whole CC tree and the precision constraint as arguments, and then returns the cheapest resolution in the list.

The `semiPropResolutions` function works similarly to the `usefulResolutions` function—first, it recursively gathers a list of resolutions for each child subtree of the root node, and then combines them together to form a list of resolutions for the whole tree, and then applies eclipse pruning to filter out useless resolutions. Additionally, the `semiPropResolutions` function uses its precision-bound argument to filter out resolutions from the final list whose precision is below the bound.

The real magic of the `semiPropResolutions` function, however, is in how it recursively gathers subresolutions: instead of calling `usefulResolutions` on each child subtree, it calls `semiPropResolutions` recursively on each child subtree using a new bound that depends on the original bound and on the type of the root node. For some kinds of root nodes, such as a combinator node with P_{sum} as a precision function, there is no useful sub-bound, so the `semiPropResolutions` simply calls itself recursively with a bound of zero (which implies no filtering, since every resolution will have a non-negative precision). However, many other node types are more forgiving—for example, for choice nodes, or for combinator nodes with P_{har} or P_{min} as a precision function, the `semiPropResolutions` function can use the original bound for the recursive calls, because in order for a resolution rooted at one of these node types to meet the bound, all of their child subresolutions must meet the same bound. For combinator nodes with $P_{\text{const}(z)}$, the `semiPropResolutions` function can divide the original bound by z when making the recursive call

to the single child.

Interestingly, not all downward-propagatable functions allow for useful sub-bounds in the recursive calls to the `semiPropResolutions` function. For example, $P_{\max}$ is a downward-propagatable function, but when the `semiPropResolutions` function is called on a tree whose root node has $P_{\max}$ as a precision function, it is forced to use zero as its new bound for the child subtrees, because an optimal resolution for that tree might involve a child subtree with zero precision (since the $P_{\max}$ function allows another subtrees precision to be used "instead").

4.3.3 Effectiveness

To evaluate the effectiveness of the `semiprop` resolver, we had it compete with the `epbrute` resolver on a series of benchmarks of downward-semi-propagatable trees, for which we expected the `semiprop` resolver to have a (slight) advantage. Unfortunately, on most benchmarks, their performances were indistinguishable. Figure 4-5 shows the resolution time for a few choice benchmarks where the `semiprop` resolver managed to (barely) edge out the `epbrute` resolver.

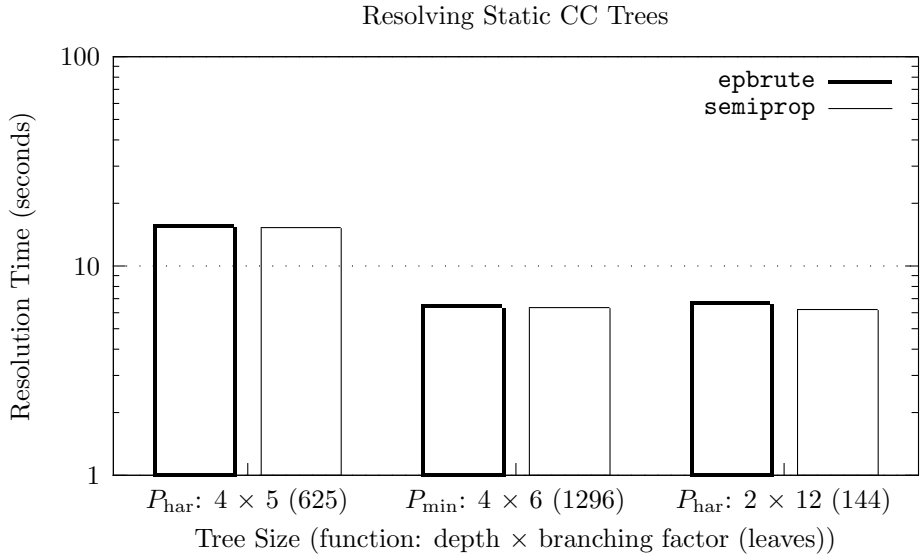

Figure 4-5: Performance comparision of the `epbrute` and `semiprop` resolvers

The upshot of these results seems to be that that while downward semi-propagation doesn't hurt, neither does it seem to help significantly when we're already using eclipse pruning, at least in practice.

4.4 Extensions to the abstraction

We previously discussed some extensions one might want to make to the (rather limited) CC trees abstraction, back in Section 2.3. Now that we have seen some techniques for resolving CC trees and have

built up an intuition for them, we're ready to look at why these extensions are troublesome, and why they are therefore beyond the scope of this thesis.

4.4.1 Program dags

We explained back in Section 2.3.1 that it would be useful if we could share inputs between different parts of the program tree, and pay the cost of a shared input just once while using its value multiple times. Figure 4-6 shows an example of what this might look like.

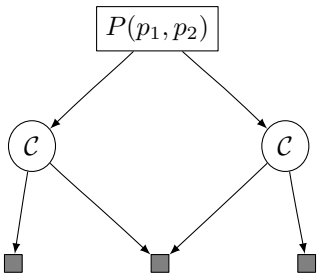

Figure 4-6: A simple CC dag

The structure of this program dag is a combinator node that combines the results from two choice nodes, each of which has a choice between two different inputs. The catch is that one input is shared between the two choice nodes, so if they both pick that one, they both get to use its precision, but the cost is paid only once.

This would be a very useful sort of structure to support, and it would be easy and natural to extend the CC tree abstraction to allow for program dags such as this one. Why, then, does Ccow not support such programs? After developing the techniques in this chapter, we are ready to discuss why this extension is so troublesome.

As we mentioned in Section 2.3.1, the fundamental issue is that because we pay the cost only once for each shared input that we use, the cost of a subtree (or rather, a subdag) is not well-defined. Consider the CC dag in Figure 4-6, and suppose that the shared input has a cost of 5, and both choice nodes choose that input. What is the cost of the left-hand choice node—is it 0, or 5, or something in between? In the CC tree abstraction, every internal node has a cost, which is equal to the sum of the costs of its children (for combinator nodes), or the cost of the chosen child (for choice nodes). By that definition, we would have to say that each choice node had a cost of 5, which would suggest that the root node had a cost of 10, yet it really has a cost of only 5.

We could solve this issue by redefining the cost of an internal node to be in terms of the set of inputs it depends on, and say that the cost set of a combinator node is the union of the cost set of its children.[8] However, this definition causes downward propagation and eclipse pruning—the two most important and

[8] In fact, in an earlier stage of this research, that is exactly what we did.

fundumental techniques developed in this chapter—to stop working!

To see this, consider the following counterexample. Consider the CC dag shown in Figure 4-6, and suppose that the shared input has precision 10 and cost 5, that the other two inputs each have precision 11 and cost 4, and that $P = P_{\min}$. If we try to resolve this CC dag, eclipse pruning tells us that we should discard the shared input at each choice node, because in both cases it is eclipsed by the other choice, leaving us with a resolution with precision 11 and cost 8. However, if our global precision constraint is 10 or less, this would be a mistake—the optimal resolution would be to pick the shared input at both choice nodes, yielding a precision of 10 for a total cost of just 5. Clearly, eclipse pruning as we know it can no longer be applied at every level of the tree when we are working with dags. Downward propagation does no better—after propagating the precision constraint of 10 downwards, the locally optimal choice at each choice node is to choose the non-shared input, again leaving us with a non-optimal resolution.[9]

When inputs are shared, we can no longer make choices in isolation, because the cost of one choice can depend on choices made elsewhere, as the above example illustrates. Without downward propagation, we are forced to resort to brute force, and without eclipse pruning, that brute force is liable to be intractable (as it was in the measurements of Section 4.1.4).

It is interesting to note, however, that although we cannot use eclipse pruning on CC dags, we *can* use downward semi-propagation to achieve some amount of pruning, because downward semi-propagation does not depend on costs.

4.4.2 Recursion/loops

We briefly discussed the possibility of extending the CC trees abstraction to allow for recursion (or loops) in Section 2.3.2, although we didn't define exactly how this would work (we speculated that either an infinite dag or a cyclic digraph might work). Dealing with such an extension would likely involve many of the problems we saw with dags above; however, with CC dags, we could still presumably use a naïve brute force approach. When recursion is permitted, even brute force may not work—the brute force approach we used for CC trees essentially involved "running" the program on the precisions instead of the inputs. When recursion is permitted, termination of the program is (presumably) not guaranteed, so this approach is likely to run into trouble regardless of how we encode recursion into the graph structure. Suffice to say, augmenting the abstraction to allow recursion would probably require a much more clever solution than the kinds of techniques we have described in this chapter, and is beyond the scope of this paper.

[9] It might seem that we should be able to apply dynamic programming for resolving CC dags, since sharing subtrees should give us overlapping subproblems—the reason we can't is that, as we see here, when we go to dags we lose the optimal substructure we had in the tree case.

4.4.3 Value-dependent precision functions

We saw in Section 2.3.4 that in some models of precision, we require the child *values* in addition to the child precisions of a combinator node in order to calculate the new precision. The fact that the CC trees abstraction does not allow for this therefore limits the kinds of precision models we can use. Why did we place this limitation?

The obvious answer would be that such an extension prevents us from performing resolution statically, since values are not known until run-time. Actually, this is not such a problem—as we'll see in the next chapter, the problem of run-time resolution is both important and soluble.

The real issue is that not only do we not know the value of a node (and therefore, the precision of a node under this extension) until run-time, but we don't know the value of a node *until we pay for inputs* that that node depends upon. In other words, we may have to pay for some inputs in order to even form a resolution. This implies that we may have to pay for an input in order to determine that we don't need actually need it, which would be quite unfortunate. There is no way around this problem, although it can be mitigated we know something in advance about the values of inputs. For example, if we knew the probability distributions from which the input values are being sampled, then we might be able to form a strategy for resolving the tree that would minimize the *expected* total cost, including the cost of inputs that we paid for but then didn't use. However, this kind of analysis is quite beyond the scope of this thesis.

4.4.4 Conditional nodes

We defined a new kind of node (other than choice and combinator nodes)—a *conditional* node—in Section 2.3.3. A conditional node is like a choice node in that it might take on the value and precision of any of its children, but we have no choice in the matter—the child is determined by a run-time predicate value.

There are two obvious ways to think about how to resolve trees containing conditional nodes. The first is to say that we must resolve the tree so that, when we run the program and the conditional branches are chosen, we must meet the precision constraint, and we must end up paying the minimal cost out of all resolutions we could have chosen. This is essentially the model we used before, without conditional nodes, but with conditional nodes this suffers the same problem that we had with value-dependent precision functions. Suppose we have a choice node with a conditional node as a child; then, we might have a situation where we must evaluate inputs that the predicate of the conditional depends on before we can know whether we should have chosen that branch of the choice node at all, and thus we might end up paying for inputs that we don't use. This model is thus beyond the scope of this paper.

The other obvious way to deal with conditional nodes is to assume that any of the children could be chosen, and when we resolve the tree, to minimize the *worst-case* cost of the tree over all possible con-

ditional branches, whild ensuring that the worst-case precision will always meet the precision constraint. This is actually not very difficult, and the Manatee Ccow compiler contains an experimental implementation of this extension, including code within the various resolution algorithms to handle conditional nodes (however, the code-generation semantics of conditional nodes have not yet been well solidified). This experimental implementation is not examined in this paper, because it turned out to add very little of interest to the discussion.

4.5 Summary

In this chapter, we've seen a handful of techniques for resolving CC trees statically, and taken some empirical measurements to evaluate how much they help. Our main conclusion is that, for static resolution, eclipse pruning makes brute force a reasonably viable technique, but downward propagation is superior when the tree allows for it. Additionally, downward semi-propagation can help in a few cases where downward propagation can't, but it doesn't seem to help all that much. We'll reexamine these techniques in the context of *partial* resolution in the next chapter.

We also described the implementation of a number of resolution algorithms supported by the Manatee Ccow compiler that use these techniques. Briefly, these resolution functions are:

Name	Section	Technique
`brute`	4.1.1	Naïve brute force
`epbrute`	4.1.2	Brute force with eclipse pruning
`downprop`	4.2.4	Downward propagation
`semiprop`	4.3.2	Downward semi-propagation

There is certainly room for other, more clever resolution algorithms; we will speculate about some of these in Chapter 7.

THIS PAGE INTENTIONALLY LEFT BLANK

Chapter 5

Partial Resolution

In the previous chapter, we examined the *static* case, where the precision and cost of each input is constant and known ahead of time. In this chapter, we consider the *dynamic* case, where some inputs may have precisions and/or costs that are not known until run-time, and which may change from run to run.

We will assume that, just before we must execute the CC tree, we can query each input to find its exact precision and cost, as described in Section 3.2.3. If we know nothing about the precision or cost of any input ahead of time, one solution would be to simply run one of our static resolution algorithms to resolve the tree at run-time, just before executing the tree, and to do this over again on each run of the program. The problem with this is that optimally resolving a CC tree can be quite expensive, as we saw in the previous chapter, and while we may be willing to pay this computation cost once at compile-time, we may not want to pay it at run-time on every run of the program. If instead, we know the precisions and costs of *most* inputs ahead of time, but have slightly incomplete information, it seems as if we should be able to *mostly* resolve the tree at compile time, leaving only a small amount of work to finish the job at run-time. This sort of *partial* resolution is the subject of this chapter.

5.1 Model

We assume that we know (at compile-time) the minimum and maximum possible precision and cost of each input, as described in Section 3.1.2. We discuss in this section why this is a reasonable model and how we can modify our approach from the previous chapter in order to deal with our only partial knowledge.

5.1.1 Motivating cases

One question we should answer before proceeding is why we should be interested in dynamic inputs as defined in Section 3.1.2. Specifically, why would we not know the precision or not know the cost of an input ahead of time? And if we don't know the precision/cost ahead of time, why would expect to nonetheless know a strict range into which it will fall (as opposed to, say, a Gaussian distribution from which it will be sampled)? Here, we'll take a quick look at a couple of practical situations in which dynamic inputs are necessary, and in which range-based dynamic inputs are sufficient.

Suppose we have an input representing a a robot's GPS sensor, which will give us the location of the robot. Furthermore, suppose that we know that the robot never moves faster than, say, 1.5 meters per second. Finally, suppose that we read the sensor to get the location of the robot to within 2 meters, and 4 seconds later, we wish to know the location of the robot. If we simply used the *same* position that we read just four seconds ago, then even without knowing what the robot has been doing since then, we know that that position value is an accurate estimate of our current position to within 8 meters. Thus, even though we know this old position is almost certainly wrong now, it may still be accurate enough for our current purposes, and moreover costs us nothing to obtain. This reuse of old values is a technique that we can use any time we have some way of bounding the precision on the reused value.

What does this look like in Ccow? Suppose that our GPS sensor is represented like this in Ccow:

```
(input gps ~0.5 $12)
```

Using the old value costs nothing, and might be just as precise as reading a new value (if we just read the sensor a moment ago), but might instead have a precision as low as zero (if the value is so old as to be completely useless). We can represent a use of the old value as an input, like this:

```
(input gpsOldValue (~0 ~0.5) $0)
```

Now, anywhere in our program where we would have used the `gps` input, we can instead choose between re-reading the input and simply using the old value:

```
(choice gps gpsOldValue)
```

Thus, range-based dynamic inputs have allowed us to capture this technique.

Now let's look at a case where dynamic *cost* is useful. Suppose we have a sensor that uses a radio, and that we wish to leave the radio turned off as much as possible. As a simplified model, turning the radio back on incurs a fixed cost, and then we must additionally pay for each message we send. Thus, using the sensor is expensive if the radio is off (since we must turn it on, and then send the message), but is cheaper if we've very recently used the sensor (since the radio will still be on, and thus we need only pay to send the message). If turning on the radio costs 10 units, and sending a message costs 2 units, then we can represent the sensor in Ccow as follows:

```
(input radio ~2 ($2 $12))
```

Of course, the range does not capture the fact that the cost will always be exactly 2 or 12 (not in between), but it's good enough. Assuming that the program has a choice between using the radio and some other alternative, if we run the program repeatedly, it will tend to avoid using the radio while the radio is off, but once it is forced to turn the radio on to meet the precision constraint, it will be able to take advantage of the radio on subsequent consecutive runs for much cheaper, until it no longer needs it and can turn it off. Thus, the natural behavior of the program will be to use the radio in bursts, turning it off in between, which is exactly what we would want.

5.1.2 Interval arithmetic

Having established that range-based dynamic precisions and costs form a reasonable model, let's now look at how to deal with that model at compile-time.

In the static case, we knew the precision and cost of each input, and by applying the precision functions at each combinator node, we could determine the precision and cost of any resolution of a CC tree (that is, the precision and cost of the tree after applying the resolution). However, in the dynamic case, rather than knowing the precision and cost of each input, we have an *interval* for the precision and cost of each input. If we apply the precision functions at each combinator node on the precision intervals of the children of that node, then we can determine the precision interval (and cost interval) of any resolution of a CC tree. Thus, for any given resolution, although we won't know until run-time what the exact precision and cost of that resolution will be, we know at compile-time the minimium and maximum precision and cost that that resolution could possibly take on.

How do we apply a precision function to intervals? Interval arithmetic on arbitrary functions can be quite tricky, as we must understand the behavior of the function within the interior of the interval [6]. However, thanks to Requirement 2.2, we know that every precision function is monotonic, which means that we can calculate the result interval using only the endpoints of the input intervals [6]. In fact, because each precision function P is monotonicly *non-decreasing*, the lower bound of the result interval is equal to the result of P when applied to the lower bounds of the input intervals, and similarly, the upper bound of the result interval is equal to the result of P when applied to the upper bounds of the input intervals. This means that performing interval arithmetic as we build up resolutions on dynamic CC trees is actually quite simple.

5.1.3 Resolution strategies

Finally, let's look at what it is that the compiler must do in the dynamic case. In the static case, the goal of the compiler is to produce a resolution of each CC tree. However, in the dynamic case, this is not possible in general—as we just saw, the precision and cost of each resolution in the dynamic case are intervals rather than single numbers, and because numeric intervals don't form a total order, we can't always know at compile time which resolution will be optimal. Thus, our goal at compile time is not to

select a resolution, but to create a *resolution strategy* that we will employ at run-time to find the optimal resolution. Having formed a strategy, we must then generate the necessary code to execute it at run-time. The next section will discribe how this is done by the Manatee Ccow compiler.

5.2 Run-time choice nodes

In order to build up resolution strategies, we introduce a new kind of node: a *run-time choice node*. A run-time choice node is supplied with a precision constraint, an acceptable cost, and a set of children. At run-time, the node makes its choice by evaluating the precisions and costs of all of its children, and choosing a child that is either *acceptable* (meets the (local) precision constraint and does not exceed the acceptable cost) or *optimal* (most cheaply meets the precision constraint). Whereas, in the static case of the previous chapter, our goal at compile time was to create a resolution (that is, a tree with choice nodes removed), our goal now is to create a *partial resolution*, or "presolution"[1]—a tree with choice nodes removed, but possibly with one or more run-time choice nodes inserted.

When we create a run-time choice node (at compile-time), we take a precision constraint, an acceptable cost, and a number of child presolutions, and join them together to form a larger presolution; in the Manatee Ccow compiler, this is done with the `formRtChoicePresolution` function. As it turns out, there are a couple of optimizations we can apply here, without knowing any of the larger context in which this run-time choice node is being inserted, to filter out some of the children at compile-time, thus reducing the work we must do at run-time. We discuss these techniques below.

5.2.1 Inadmissibility pruning

The first and most obvious optimization on run-time choice nodes is to eliminate choices which are guaranteed at compile time not to meet the local constraint (that is, child presolutions whose upper bounds on precision are strictly less than the local precision constraint). We would never be able to pick such a choice at run-time, so we may as well eliminate them at compile-time. As the Manatee Ccow compiler builds up presolutions in the IR, it annotates each presolution with its precision and cost ranges, so this filtering is easy to perform.

5.2.2 Acceptable/optimal selection

A second optimization is to look through the list of child presolutions of the run-time choice and check to see if one exists that is 1) guaranteed to be admissible (that is, the lower bound on precision is greater than or equal to the local precision constraint), and 2) guaranteed to be acceptable (that is, the upper bound on cost is less than or equal to the local acceptable cost). If such a child presolution exists, then we

[1]The term "presolution" is something of a pun. It is short for "partial resolution," but the name also suggests that it is a "pre-solution"—an intermediate result that comes before the actual solution, which is of course the full resolution that we determine at run-time.

can choose this child at compile-time, without any need to query its actual precision or cost at run-time; this is called *acceptable selection*. It is possible, of course, that we will only very rarely encounter such a choice, but this optimization is very much worth trying, since it is cheap to check (linear time in the number of children of each run-time choice node), and may occasionally save a significant amount of work at run-time.

To account for the possibility that the optimal resolution might not be acceptable, we can check for one other case: if the list of child presolutions includes one that is definitely admissible and definitely cheaper than all other child presolutions (that is, its upper bound on cost is less than or equal to the lower bound on cost of all other child presolutions), then we choose this child at compile-time; this is called *optimal selection*.

This technique is critical for allowing us to resolve static subtrees at compile-time. If we have a subtree, rooted at a run-time choice node, that contains a number of dynamic inputs, then of course we will likely have to resolve the run-time choice at run-time. However, if we have some other subtree, also rooted at a run-time choice node, that contains only static inputs, then by rights we ought to be able to resolve that subtree at compile-time. Since static inputs lead to subtrees with point-like precision and cost ranges (that is, ranges where the upper and lower bounds are equal), acceptable and optimal selection subsumes static resolution of static subtrees—if all children of a particular run-time choice node have point-like precisions and costs, then there *will* be one child that is definitely cheaper than the others. Thus, as long as we can produce local run-time choice nodes, with local constraints, we can statically resolve static portions of the dynamic CC tree—we'll see this happen in Section 5.4.

5.2.3 Search ordering

Even if we can't do optimal selection at a particular run-time choice node, we can still order the run-time search so that we will sometimes be able to stop early. For example, suppose that we have a run-time choice node with six children; the first three children have overlapping cost intervals, as do the last three children, but the first three children are definitely cheaper than the last three (that is, the maximum of the upper bounds on cost for the first three children is less than or equal to the minimum of the lower bounds on cost for the last three children). At run-time, we can check the first three children first, querying their actual precision and cost. If at least one of them is feasible, than the cheapest of those is the optimal choice, because it is both feasible and definitely cheaper than the last three children—we need to query the actual precision and cost of the last three children only if none of the first three turns out to be feasible.

We can generalize this technique as follows. Given the list of child presolutions for a run-time choice node, we first sort the list (at compile-time) by the lower bounds on cost. At run-time, we examine each child in the list in order, keeping track of the cheapest feasible child we've seen so far. As soon as we reach a child whose lower bound on cost is greater than or equal to the cheapest feasible child we've

found, we know that all remaining children in the list will be at least as expensive as the one we've found, so we need not examine them.

This technique can potentially save a great deal of time at run-time—without it, we would have to examine *all* children of a run-time choice node, and pick the cheapest feasible child out of all of them.

5.2.4 Run-time implementation

Finally, we turn to the question of how run-time choice nodes are to be implemented by the Code Generation phase of the compiler. Once the list of children has been pruned and ordered at compile-time, the basic steps that must be performed at run-time are as follows.

1. In the Code Generation phase, in addition to the whole presolution tree, we generate a *choice tree*—a tree of only run-time choice nodes, created by removing all other nodes from the presolution tree.

2. At run-time, we resolve the choice tree recursively from the top down.

   - At each run-time choice node, we recursively resolve each child subtree, in order, to produce a `resolution_t` object, which stores the precision and cost of the resolution, as well as a representation of which choices were made.
   - As we go, we keep track of the cheapest feasible resolution we have seen so far. When we reach a child whose lower bound on cost is greater than or equal to the cost of our feasible resolution, we terminate the search.
   - Once we have chosen a child, we return a `resolution_t` object representing the child we chose and the subresolution rooted at that child.

3. Having obtained the top-level `resolution_t` object, we can then execute the presolution tree, using the `resolution_t` object to tell us which choices to make.

If the choice tree contains n nodes and has a maximum branching factor of k, it should be clear that this algorithm will require at most $O(kn)$ work at run-time, which is not unreasonable, especially for small branching factors.

5.3 Brute force partial resolution

Now that we have the run-time choice primitive, we are ready to develop some partial resolution algorithms. What we will find is that run-time choices allow us to develop algorithms that are nearly identical to the algorithms described in Chapter 4, even though those algorithms were designed with the simpler, static case in mind.

5.3.1 Straightforward brute force

First, let us consider the most obvious, straw-man resolution algorithm. This algorithm can be simply described in steps as follows:

1. Generate a list of every possible resolution of the given program tree.
2. Create a run-time choice node $\mathcal{C}$ with these resolved trees as children, and with the top-level precision constraint and acceptable cost.
3. Return the resulting tree rooted at $\mathcal{C}$.

We then pass this presolution tree to the Code Generate phase, which generates code as described in Section 5.2.4. It should be clear that this scheme will indeed allow us to find the optimal resolution at run-time, because we are allowing the run-time choice node to choose between every possible resolution. Unfortunately, as in the analogous static brute force algorithm described in Section 4.1.1, the number of possible resolutions is in general, exponential. This means that the run-time resolution time, and even the size of the resulting executable, are liable to be exponential in the number of choices in the original tree. This is bad news.

The `brute` resolution function included in the Manatee Ccow compiler will use the above strategy to partially resolve CC trees when given a program containing dynamic inputs.

5.3.2 Eclipse pruning

In the static case, we were able to improve the naïve brute force algorithm by applying *eclipse pruning*, which we proved in Section 4.1.2 we could safely apply at every level of the tree, from the bottom up. We can do the same thing in the dynamic case, with one snag: because we have intervals for precision and cost instead of exact values, we no longer always know for sure at compile-time whether or not one particular resolution will eclipse another. We can safely eliminate a resolution R_2 only if we are *sure* it will be eclipsed by another resolution R_1—that is, if the lower bound on precision for R_1 is greater than or equal to the upper bound on precision for R_2, and if the upper bound on cost for R_1 is less than or equal to the lower bound on cost for R_2.

The algorithm from Section 4.1.2 used a clever trick that allowed us to perform eclipse pruning on a list of resolutions in linear time after sorting. However, because we have intervals for precision and cost instead of exact values, we can no longer sort a list of resolutions by precision (we could sort by, say, the lower bound on precision, but this would not be enough to allow the clever trick to work). Thus, we have to resort to a quadratic-time algorithm for applying eclipse pruning to a list of resolutions: we simply compare each pair or resolutions, and remove one if it will definitely be eclipsed by the other.

The `epbrute` resolution function included in the Manatee Ccow compiler will use this strategy when given a program containing dynamic inputs—like the dynamic `brute` resolver, it constructs a list of

top-level resolutions and joins them with a run-time choice node, but like the static `epbrute` resolver, it applies this weakened eclipse pruning at each level as it builds up the list of top-level resolutions.

We might be discouraged by the fact that in the dynamic case, eclipse pruning is both slightly more expensive algorithmically and potentially less effective, but in fact, we have a huge advantage here over the static case. For static resolution, eclipse pruning is merely an optimization on compile-time—a trick to try to make the brute-force compilation faster. In this case, however, eclipse pruning allows us to trade compile-time speed for run-time speed—by paying for eclipse pruning once, at compile-time, we make every run of the program faster as compared to programs compiled by the `brute` resolver. Thus, in this case, eclipse pruning is much closer to what one would normally think of as a compiler optimization.

5.3.3 Downward semi-propagation

Now that we have seen how to adapt the `brute` and `epbrute` resolvers to deal with dynamic CC trees, it is a simple step to make the same kind of adaptation for the `semiprop` resolver. When operating on CC trees with dynamic inputs, the `semiprop` resolver uses the `semiPropPresolutions` function to gather presolutions of the tree. This function works just like the `semiPropResolutions` function that we used in the static case, except that it filters out only presolutions that *definitely* will not meet the local bound. After collecting all the presolutions together, the `semiprop` resolver joins them with a run-time choice node, just as it did for the brute force partial resolvers above.

5.3.4 Effectiveness

To test the effectiveness of the partial resolvers in this chapter, we constructed a number of benchmark Ccow programs using dynamic precisions and costs, compiled them with the Manatee Ccow compiler using various resolution functions, and recorded the number of run-time options (that is, children of run-time choice nodes) that would need to be considered at run-time when executing the program. Since the work that must be done at run-time is proportional to the number of these options, this gives us an objective measure of the effectiveness of different partial resolvers. As with the static benchmarks, we did not have room to include the full source code for every benchmark, but the source code for one of our partial-resolution benchmarks is given in Appendix B.

For this batch of measurements, we construct a batch of benchmarks of various sizes using $P_{\min}$ combinators, and ran the the `brute`, `epbrute`, and `semiprop` resolvers on each one. The results are shown in Figure 5-1. As it turned out, the results for the `epbrute` and `semiprop` resolvers were exactly the same on every benchmarks—the `semiprop` resolver was unable to eliminate any options that the `epbrute` couldn't already eliminate—so we have not bothered to show the results for the `semiprop` resolver in the figure.

Note that this benchmark set includes two different trees of depth 4 and branching factor 3, with different amounts of tightness in the input precision and cost ranges (the one with the higher number of

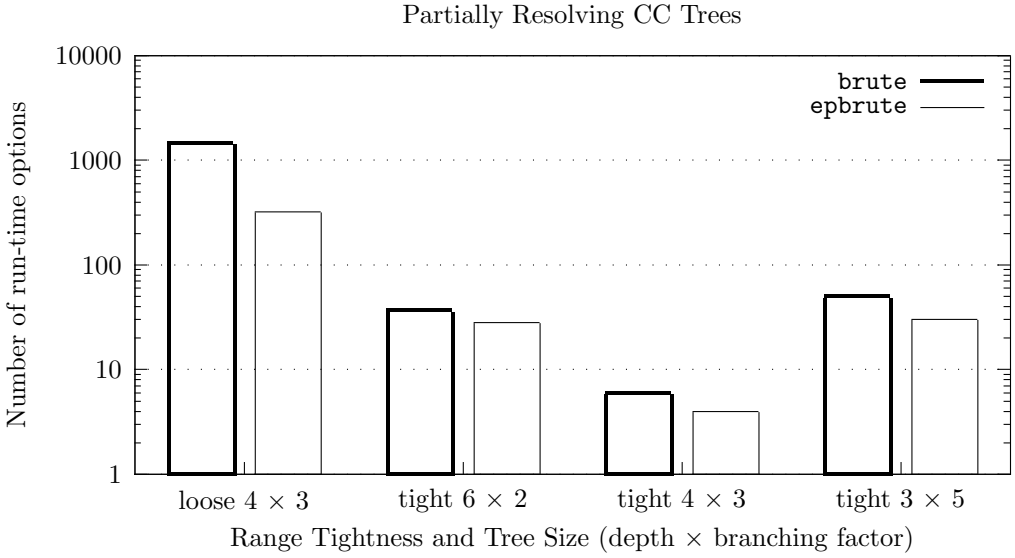

Figure 5-1: Comparision of the `brute` and `epbrute` partial resolvers

run-time options is, of course, the one that had looser ranges).

The results indicate that eclipse pruning is still a significant improvement over naïve brute force, and that downward semi-propagation is little, if any, improvement over eclipse pruning, which was also true in the static case of the previous chapter. However, we note that in the static case, eclipse pruning yielded a couple orders of magnitude improvement over naïve brute force in compilation time, whereas here, the run-time improvement is more modest. This makes sense—with less knowledge about precisions and costs here, we are less often able to safely eclipse-prune presolutions.

Finally, we note that in some cases, the number of run-time options that must be considered for even a small tree using these resolvers can be enormous. We would prefer to find a more effective way of reducing the amount of work that must be performed at run-time.

5.4 Downward propagation

Next, we reexamine the *downward propagation* technique we developed in Section 4.2. Downward propagation allowed us to resolve CC trees much more quickly when we were fortunate enough to encounter certain kinds of precision functions in combinator nodes. However, in the static case, it relied on resolving subtrees optimally before resolving larger trees. In the dynamic case, we may not be able to statically resolve a subtree optimally, so it might seem that we won't be able to use downward propagation for CC trees with dynamic inputs.

The key is to remember that downward propagation allows us to propagate the precision constraint downwards, so that it becomes a local constraint rather than a global constraint. As long as we have

a local constraint, we can use a run-time choice node locally, rather than at the top level as we did for brute-force partial resolution. If we create an algorithm analogous to the static downward propagation algorithm that builds up a partially resolved tree rather than a fully resolved tree, using run-time choice nodes to encode decisions that cannot be made statically, we will find that the resulting tree essentially encodes the downward propagation strategy, which allows run-time resolution to proceed much more quickly than when we use the `brute` or `epbrute` resolvers. However, this downward propagation scheme wins in an even bigger way: by making constraints local, it allows static portions of the tree to be resolved statically via optimal selection (as described in Section 5.2.2), so that if a tree is mostly static, we need do only a small amount of work at run-time to finish the resolution.

5.4.1 Implementation

When using the `downprop` resolver on a tree containing dynamic inputs, the Manatee Ccow compiler uses an alternate algorithm. This new function is called `presolveDownProp`, and works very similarly to the `resolveDownProp` function used by the `downprop` resolver in the static case—the main difference is that occasionally we must insert a run-time choice node into the tree we are building. When called, the `presolveDownProp` function examines the root node R of the given tree T, and behaves as follows:

- If R is a leaf (that is, T contains just one node), then return the trivial presolution of T.
- If R is a choice node, then call `presolveDownProp` recursively on each child with the same precision constraint and acceptable cost; take the resolutions for those children for which a feasible subresolution exists, and join them with a run-time choice node using the local precision constraint and acceptable cost.
- If R is a combinator node, then examine the precision function P:
  - If $P = P_{\text{const}(z)}$, then call `presolveDownProp` recursively on each child with a precision constraint of zero and an acceptable cost of zero, and join the resulting subpresolutions together. No run-time choice is needed.
  - If $P = P_{\text{scale}(z)}$, then there is exactly one child; call `presolveDownProp` recursively on this child with the same acceptable cost, and a precision constraint of s/z, where s is the original precision constraint.
  - If $P = P_{\min}$, then call `presolveDownProp` recursively on each child with the same precision constraint and an acceptable cost of zero, and join the resulting subpresolutions together. No run-time choice is needed.
  - If $P = P_{\max}$, then things get messy. Call `presolveDownProp` recursively on each child twice—once with the same precision constraint and an acceptable cost of zero, to get an optimal subresolution, and once with a precision constraint of zero (as in the const case), to get the

cheapest subresolution. For each child, consider the presolution formed by joining that child's optimal subpresolution the cheap subresolutions of the other children. Join all these composite presolutions together with a run-time choice node.

- If $P = P_{\text{har}}$, use the `semiprop` resolver to resolve T.
- If P is any other function, use the `epbrute` resolver to resolve T.

It should not be surprising that we used a run-time choice node for choice nodes, but it is interesting that we must also use a run-time choice node for $P_{\max}$ combinator nodes. In this situation as well as that of Section 4.3.2, the $P_{\max}$ function is rather stickier than most of its fellow downward-propagatable functions.

This algorithm is considerably more effective than the other partial resolution algorithms, because it creates many local run-time choice nodes rather than a single, top-level run-time choice node. Any local run-time choice node whose subtree is static will always be resolved statically, so if large portions of the tree are static, creating many local run-time choice nodes will allow us to do most of the resolution statically.

5.4.2 Effectiveness

To evaluate the effectiveness of the `downprop` resolver, we had it compete with the `epbrute` resolver on a series of benchmarks of downward-propagatable trees, of various sizes, and with differing degrees of tightness in the precision and cost ranges of the inputs. Again, the number we measured was the total number of run-time options, since the run-time resolution take time proportional to this number. The `epbrute` resolver puts all options into a single run-time choice node at the top of the tree, so its number is simply the number of children of this one run-time choice node. The `downprop` resolver, on the other hand, distributes run-time choice nodes throughout the tree, so its number is the sum of the numbers of children of all these nodes. The results are shown in Figure 5-2.

These results were quite a surprise. Although the `downprop` resolver beat the `epbrute` resolver thoroughly on benchmarks with loose ranges, the `epbrute` beat the `downprop` resolver on benchmarks with tight ranges, sometimes enormously! Also, the `epbrute` resolver's performance varied wildly across benchmarks, while the `downprop` resolver performed quite evenly across the various benchmarks of roughly the same size.

These results reveal the fundamental difference between these two resolvers. The `downprop` resolver works by making local choices on local constraints, and doesn't much care about the tightness of the input ranges—we need only to be able to propagate constraints downward. The `epbrute` resolver, on the other hand, works by considering resolutions globally, and eliminating provably inferior resolutions. In some cases, taking this global view allows it to reduce the number of choices more than the `downprop` resolver, but in other cases, it becomes overwhelmed with large numbers of unprunable resolutions.

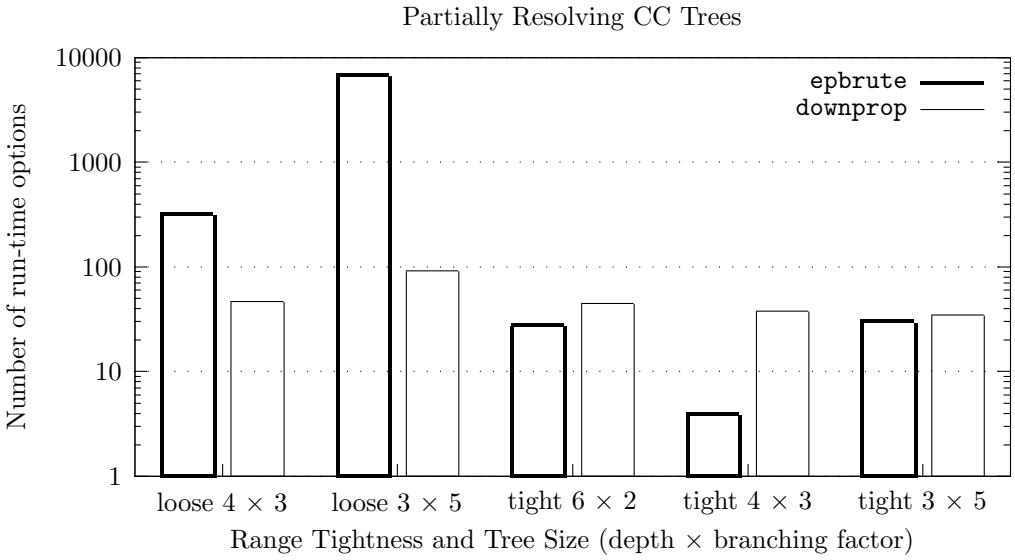

Figure 5-2: Comparision of the `epbrute` and `downprop` partial resolvers

5.5 Summary

In this chapter, we've seen a handful of techniques for partially resolving dynamic CC trees, related them to the analogous techniques in the static case, and taken some measurements to evaluate how much they help. Our main conclusions are that, for partial resolution, eclipse pruning is a substantial improvement over naïve brute force, and that downward propagation is superior when the tree allows for it *and* when we have only loose bounds on the inputs, but that eclipse-pruning brute force may sometimes be able to do better. Perhaps most important, however, is that these techniques show that we *can* resolve trees partially, in a way that significantly reduces the work we must perform at run-time, compared to simply running a static resolution algorithm at run-time.

Chapter 6

Related Work

6.1 Precision/cost tradeoffs in data gathering

Although researchers have identified a need for handling cost and precision tradeoffs, there is no commonly used framework for programming with uncertain inputs. Sugihara and Gupta's investigation of programming models for sensor networks, an area where cost constraints have driven research for systems to manage precision/cost tradeoffs, reveals that current systems provide only limited frameworks for describing the quality of data [19].

Sugihara and Chien present an "accuracy-aware" system for managing resource usage in a sensor network based on the assumption that they can model groups of sensors with linear regressions [17]. They provide a compact, sufficiently accurate data representation by combining regression planes summarizing the error bound on their data points and explicit representations for remaining points. Sugihara and Chien also provide a distributed algorithm for cost-effective data gathering based on this representation of the data [18].

The MiLAN (Middleware Linking Applications and Networks) system for sensor networks takes a description of application requirements in order to monitor network conditions and optimize sensor and network configurations [7]. MiLAN has a concept of "quality of service" (QoS) expressed by a number from 0.0 to 1.0. Given that the programmer declares specification of the variables of interest, the required QoS of each variable, and the level of QoS for each set of sensors, MiLAN chooses the set of sensors that best meet the cost vs. QoS tradeoff. On the surface, this sounds very similar to Ccow, but their goal is actually rather different—MiLAN determines which subset of a set of sensors to aggregate together in a uniform way, whereas Ccow is intended to allow *arbitrary* choices in *arbitrary* points in a program.

There have also been sensor network systems based on expressing data inputs probabilistically [19]. For example, the BBQ framework is a model-based query processor for TinyDB that provides a statistical guarantee on the accuracy of the data in terms of confidence levels, but it is based on the assumption

that the data follows a multivariate Gaussian distribution. Compressive Wireless Sensing guarantees the accuracy of the data without prior knowledge of data priorites but assumes Gaussian noise.

The SARANA system provides a prototype language, compiler, and run-time system for providing support for programmers to handle geographic regions of interest and to express policies about tradeoffs involving the quality of the result [5]. This work shows that cost and precision tradeoffs are an issue in mobile networks as well and that research in languages that support these tradeoffs is important; however, SARANA is somewhat specific to sensor networks, and like MiLAN it allows for only a vague notion of quality rather than a precise description of precision.

There has also been work on propagating uncertainty decoupled from physical systems. Stanford's Trio project is a database management system based on uncertain inputs [2]. They present an extension of relational databases with extensions for managing uncertainty. Their notion of an uncertain database is one that represents multiple intances of the possible state of the database.

6.2 Probabilistic robotics programming

One of the early target applications of the Cbass project was to aid researchers working with autonomous underwater vehicles (AUVs). The length of an AUV mission is limited by resource constraints; in particular, battery life is usually a limiting factor. Although costs and precisions are mostly known in advance, the system has no way of adapting to precision requirements. Instead, the system queries all sensors rapidly and continuously to achieve the highest possible precision in its calculations, which is usually higher than is actually necessary. This over-sampling increases the battery usage of the system, and therefore places an additional limitation on mission duration. AUVs typically have different precision requirements under different conditions—for example, surveillance requires higher precision than navigation in the open sea—so it would help them to have a system for minimizing the battery cost for the required precision at any point in time.

The current approaches to probabilistic robotics programming that other researchers describe also ignore extra cost incurred from over-sampling [1, 20, 22].

6.3 Program sketching

The "choice" model used in Ccow bears some resemblance to *program sketching*, as described by Solar-Lezama et al.; they describe a language called Sketch that allows programmers to write a partial program, with *holes* to be filled in by the compiler, and provide a specification of correct program behavior, from which the compiler derives the missing pieces of the program [16]. Ccow, by contrast, requires the programmer to provide the compiler with concrete choices, between which the compiler then chooses. However, Ccow and Sketch are solving two different problems: Sketch endeavors to find a

correct program that is *reasonably* computationally efficient, whereas Ccow endevors to find a *sufficiently precise* approximation program that *minimizes* some kind of abstract cost.

6.4 Stochastic programming languages

As we saw in previous chapters, one application of Ccow is in representing probability distributions within a program, using node precision to represent, say, the variance of the distribution. There has been much work in other languages and libraries for representing probability distributions. The Church programming language and the Probabilistic Scheme Library provide implementations of language constructs for querying, sampling, and expectation [4, 14]. Koller, McAllester, and Pfeffer present a first-order functional language with a "coin flip" construct [8]. Pfeffer generalizes this construct to create distributions over multiple possibilities [12]. Ramsey and Pfeffer show that one can efficiently implement expectation sampling support over probability distributions using the framework of the probability monad [15].

There has also been work in robotics-specific stochastic programming languages. Thrun has presented a C++-like extension for robotics that has two main contributions: it makes probability distributions useable as floating-point numbers and allows the smooth integration of function approximators (i.e., neural nets) into the language [21]. Their approach is different from that of this work, however, because it focuses on making learning algorithms easier to implement, and ignores the issue of cost.

Park, Pfenning, and Thrun present $\lambda_{\bigcirc}$, an OCaml variant that supports sampling for probability distributions [11]. $\lambda_{\bigcirc}$ provides support for expectation, variance, and conditional probabilities. They describe applications in robotic localization, people tracking, and robotic mapping (constructing a state map).

While this work highlights the importance of dealing with probability distributions in robotics programming and in other applications, none of it addresses the issue of minimizing cost while meeting precision requirements, which is the fundamental goal of Ccow. It would seem that this issue has not yet been much explored in the field of stochastic programming languages.

THIS PAGE INTENTIONALLY LEFT BLANK

Chapter 7

Conclusions

We have developed a solid core for the general problem of making precision/cost tradeoffs in programs. The CC trees abstraction, while limited, yields a difficult resolution problem, for which we have developed a number of solution techniques. We have also designed a simple programming language, Ccow, for dealing with this abstraction and have created a practical implementation, Manatee, of that language. Using this implementation, we found that for resolving static CC trees, downward propagation, downward semi-propagation, and eclipse pruning were powerful and effective tools for mitigating the potentially exponential cost of finding an optimal resolution—in particular, on our benchmarks, using eclipse pruning improved resolution time by almost two orders of magnitude over naïve brute force, and downward propagation improved resolution time over eclipse-pruned brute force by an order of magnitude for trees of medium branching factor, and two orders of magnitude for trees of large branching factor. For dynamic CC trees, where we must perform partial resolution at compile-time, we found that downward propagation and eclipse pruning were both quite effective—offering a significant improvement over a naïve brute force method—and that each was the more effective technique on different benchmarks—downward propagation beat out eclipse-pruned brute force by one to two orders of magnitude on trees with loose precision/cost ranges, while eclipse-pruned brute force edged out downward propagation by up to an order of magnitude on trees with tight precision/cost ranges.

While we have laid a foundation in this paper, there is much room for future work. We described a number of useful extensions to the CC trees abstraction in Section 2.3, and discussed why they were beyond the scope of this work in Section 4.4. However, many of the difficulties with these extensions are likely solvable with more clever techniques, and solving them could lead to programming languages that are much more useful and practical than Ccow.

Even within the CC trees abstraction, there is likely room for more clever resolutions algorithms, especially for partial resolution. In this paper we explored partial resolution strategies based on run-time choice nodes; these turned out to be a powerful building-block, but they are by no means the only way

to form partial resolutions, and other methods might yield better resolvers that require even less work at run-time. Another useful improvement would be to implement an *incremental* resolver—one which would start by finding a feasible, but costly, resolution, and then incrementally improve the resolution by cheapening it; this would allow us to devote variable amounts of time to the resolution process, and get an answer whose goodness was proportional to the amount of time we spent searching for it.

It is our hope that future work will be able to build upon this paper to build practical programming languages and models for dealing with precision/cost tradeoffs in the real world.

Appendix A

Ccow BNF Grammar

Below we include the complete BNF grammar for the Ccow language. Note that ::= is used for usual grammatical rules, while $\sim$ indicates that the non-terminal on the left is defined by the regular expression on the right.

⟨program⟩ ::= ⟨declaration⟩$^+$

⟨declaration⟩ ::= ⟨comb-declaration⟩
| ⟨input-declaration⟩
| ⟨tree-declaration⟩

⟨comb-declaration⟩ ::= "`(`" "`comb`" ⟨identifier⟩ ⟨comb-def⟩ "`)`"

⟨comb-def⟩ ::= "`scale`" ⟨decimal⟩
| ⟨comb-multi-arity-def⟩ ⟨integer⟩

⟨comb-multi-arity-def⟩ ::= "`constant`" ⟨prec-literal⟩
| "`harmonic`"
| "`hash`" ⟨identifier⟩
| "`maximum`"
| "`minimum`"
| "`sum`"

⟨input-declaration⟩ ::= "`(`" "`input`" ⟨identifier⟩ ⟨input-prec-spec⟩ ⟨input-cost-spec⟩ "`)`"

⟨input-prec-spec⟩ ::= ⟨prec-literal⟩
| "`(`" ⟨prec-literal⟩ ⟨prec-literal⟩ "`)`"

⟨input-cost-spec⟩ ::= ⟨cost-literal⟩
| "`(`" ⟨cost-literal⟩ ⟨cost-literal⟩ "`)`"

⟨tree-declaration⟩ ::= "`(`" "`tree`" ⟨identifier⟩ ⟨prec-literal⟩ ⟨cost-literal⟩ ⟨expression⟩ "`)`"

⟨expression⟩ ::= ⟨identifier⟩
| ⟨choice-expression⟩
| ⟨comb-expression⟩

⟨choice-expression⟩ ::= "`(`" "`choice`" ⟨expression⟩$^+$ "`)`"

⟨comb-expression⟩ ::= "`(`" ⟨identifier⟩ ⟨expression⟩$^+$ "`)`"

⟨integer⟩ ∼ `[0-9]+`

⟨decimal⟩ ∼ `[0-9]+(\.[0-9]+)?`

⟨prec-literal⟩ ∼ `~[0-9]+(\.[0-9]+)?`

⟨cost-literal⟩ ∼ `\$[0-9]+(\.[0-9]+)?`

⟨identifier⟩ ∼ `[A-Za-z][A-Za-z0-9]*`

The ⟨identifier⟩ non-terminal carries the additional restriction that an identifier cannot be any of the following reserved words:

```
choice      maximum
comb        minimum
constant    scale
harmonic    sum
hash        tree
input
```

Appendix B

Example Ccow Benchmarks

Here we include Ccow source code for two of the benchmarks that were used to measure our Ccow implementation: one benchmark for static resolution, and one benchmark for partial resolution (we would have included source code for all of our benchmarks, but that would have required a prohibitive amount of space).

Our benchmarks were generated by a script that, given a brief specification of the overall structure and size of the desired tree, generates a Ccow program for a random CC tree and chooses a precision constraint that is possible, but nontrivial, to meet. The source code given below has been adjusted to be more readable, but is otherwise the same code used in our benchmark suite.

B.1 Example static-resolution benchmark

The following is one benchmark that was used to compare the effectiveness of various static resolution algorithms; it is the benchmark represented by the first column of Figure 4-2.

```
(comb comb0 hash geomean 2)   (comb comb1 hash geomean 2)
(comb comb2 hash geomean 2)   (comb comb3 hash geomean 2)
(comb comb4 hash geomean 2)   (comb comb5 hash geomean 2)
(comb comb6 hash geomean 2)   (comb comb7 hash geomean 2)
(comb comb8 hash geomean 2)   (comb comb9 hash geomean 2)
(comb comb10 hash geomean 2)  (comb comb11 hash geomean 2)
(comb comb12 hash geomean 2)  (comb comb13 hash geomean 2)
(comb comb14 hash geomean 2)  (comb comb15 hash geomean 2)
(comb comb16 hash geomean 2)  (comb comb17 hash geomean 2)
(comb comb18 hash geomean 2)  (comb comb19 hash geomean 2)
(comb comb20 hash geomean 2)
(input input0 ~5.98 $79.00)    (input input1 ~3.45 $27.00)
(input input2 ~8.39 $14.00)    (input input3 ~4.11 $29.00)
(input input4 ~6.13 $58.00)    (input input5 ~8.05 $11.00)
(input input6 ~6.65 $49.00)    (input input7 ~2.96 $86.00)
(input input8 ~7.92 $45.00)    (input input9 ~1.33 $33.00)
(input input10 ~3.08 $29.00)   (input input11 ~4.43 $30.00)
```

```
(input input12 ~7.89 $25.00)   (input input13 ~7.86 $57.00)
(input input14 ~7.05 $97.00)   (input input15 ~9.87 $15.00)
(input input16 ~7.74 $46.00)   (input input17 ~8.48 $63.00)
(input input18 ~10.03 $25.00)  (input input19 ~5.82 $90.00)
(input input20 ~5.80 $37.00)   (input input21 ~6.98 $44.00)
(input input22 ~7.38 $56.00)   (input input23 ~5.41 $25.00)
(input input24 ~4.73 $51.00)   (input input25 ~8.77 $62.00)
(input input26 ~9.22 $47.00)   (input input27 ~7.46 $49.00)
(input input28 ~7.59 $50.00)   (input input29 ~10.64 $100.00)
(input input30 ~3.71 $31.00)   (input input31 ~9.95 $11.00)
(input input32 ~6.31 $89.00)   (input input33 ~8.99 $19.00)
(input input34 ~9.60 $28.00)   (input input35 ~4.51 $51.00)
(input input36 ~1.19 $94.00)   (input input37 ~2.05 $38.00)
(input input38 ~1.54 $53.00)   (input input39 ~6.24 $84.00)
(input input40 ~6.08 $71.00)   (input input41 ~10.65 $37.00)
(input input42 ~8.15 $79.00)   (input input43 ~9.36 $10.00)
(input input44 ~7.05 $14.00)   (input input45 ~8.42 $21.00)
(input input46 ~3.16 $18.00)   (input input47 ~7.25 $73.00)
(input input48 ~6.56 $68.00)   (input input49 ~6.50 $72.00)
(input input50 ~1.24 $80.00)   (input input51 ~8.58 $58.00)
(input input52 ~10.26 $38.00)  (input input53 ~10.08 $84.00)
(input input54 ~6.84 $57.00)   (input input55 ~10.86 $47.00)
(input input56 ~4.50 $99.00)   (input input57 ~8.53 $72.00)
(input input58 ~8.71 $85.00)   (input input59 ~7.64 $10.00)
(input input60 ~2.63 $10.00)   (input input61 ~10.99 $16.00)
(input input62 ~2.62 $76.00)   (input input63 ~3.85 $43.00)
(tree program ~3.81 $0.00
  (comb20 (choice (comb4  (choice (comb0  (choice input0  input1)
                                          (choice input2  input3))
                                  (comb1  (choice input4  input5)
                                          (choice input6  input7)))
                          (choice (comb2  (choice input8  input9)
                                          (choice input10 input11))
                                  (comb3  (choice input12 input13)
                                          (choice input14 input15))))
                  (comb9  (choice (comb5  (choice input16 input17)
                                          (choice input18 input19))
                                  (comb6  (choice input20 input21)
                                          (choice input22 input23)))
                          (choice (comb7  (choice input24 input25)
                                          (choice input26 input27))
                                  (comb8  (choice input28 input29)
                                          (choice input30 input31)))))
          (choice (comb14 (choice (comb10 (choice input32 input33)
                                          (choice input34 input35))
                                  (comb11 (choice input36 input37)
                                          (choice input38 input39)))
                          (choice (comb12 (choice input40 input41)
                                          (choice input42 input43))
                                  (comb13 (choice input44 input45)
                                          (choice input46 input47))))
                  (comb19 (choice (comb15 (choice input48 input49)
                                          (choice input50 input51))
                                  (comb16 (choice input52 input53)
```

```
                                    (choice input54 input55)))
                          (choice (comb17 (choice input56 input57)
                                          (choice input58 input59))
                                  (comb18 (choice input60 input61)
                                          (choice input62 input63)))))))
```

B.2 Example partial-resolution benchmark

The following is one benchmark that was used to compare the effectiveness of various partial resolution algorithms; it is the benchmark represented by the first column of Figure 5-1.

```
(comb comb0 minimum 3)  (comb comb1 minimum 3)
(comb comb2 minimum 3)  (comb comb3 minimum 3)
(comb comb4 minimum 3)  (comb comb5 minimum 3)
(comb comb6 minimum 3)  (comb comb7 minimum 3)
(comb comb8 minimum 3)  (comb comb9 minimum 3)
(input input0 (~4.52 ~7.38) ($77.67 $84.33))
(input input1 (~9.33 ~10.62) ($51.00 $77.00))
(input input2 (~2.00 ~2.38) ($23.39 $32.61))
(input input3 (~2.69 ~2.99) ($65.00 $73.00))
(input input4 (~3.21 ~3.21) ($41.30 $46.70))
(input input5 (~1.87 ~2.17) ($61.35 $62.65))
(input input6 (~8.65 ~9.23) ($76.49 $85.51))
(input input7 (~1.86 ~3.00) ($37.03 $48.97))
(input input8 (~6.05 ~8.65) ($73.53 $122.47))
(input input9 (~3.75 ~6.20) ($96.16 $103.84))
(input input10 (~9.78 ~10.78) ($17.96 $24.04))
(input input11 (~6.07 ~8.33) ($27.76 $38.24))
(input input12 (~6.97 ~9.16) ($31.16 $38.84))
(input input13 (~8.40 ~8.71) ($71.96 $76.04))
(input input14 (~5.66 ~8.97) ($69.32 $108.68))
(input input15 (~2.85 ~3.19) ($85.00 $91.00))
(input input16 (~7.39 ~8.09) ($81.99 $96.01))
(input input17 (~1.53 ~2.31) ($43.05 $52.95))
(input input18 (~0.97 ~1.07) ($44.89 $55.11))
(input input19 (~6.90 ~7.44) ($67.81 $90.19))
(input input20 (~5.75 ~7.48) ($31.76 $40.24))
(input input21 (~0.81 ~1.33) ($76.56 $121.44))
(input input22 (~5.79 ~5.89) ($18.07 $25.93))
(input input23 (~3.11 ~4.77) ($21.40 $24.60))
(input input24 (~10.88 ~10.93) ($74.30 $77.70))
(input input25 (~1.07 ~1.47) ($28.18 $39.82))
(input input26 (~3.07 ~4.17) ($65.83 $108.17))
(input input27 (~5.59 ~7.42) ($38.15 $41.85))
(input input28 (~5.98 ~7.75) ($40.41 $57.59))
(input input29 (~8.73 ~9.84) ($63.47 $96.53))
(input input30 (~8.63 ~9.44) ($61.95 $72.05))
(input input31 (~8.22 ~9.23) ($64.47 $89.53))
(input input32 (~7.00 ~10.99) ($13.12 $16.88))
(input input33 (~8.08 ~9.10) ($46.89 $47.11))
(input input34 (~6.58 ~8.61) ($49.29 $50.71))
(input input35 (~4.45 ~4.55) ($11.10 $12.90))
```

```
(input input36 (~8.41 ~9.50) ($46.85 $51.15))
(input input37 (~3.84 ~4.63) ($60.96 $81.04))
(input input38 (~6.35 ~10.40) ($40.86 $57.14))
(input input39 (~5.24 ~7.22) ($77.27 $116.73))
(input input40 (~7.92 ~9.88) ($52.91 $75.09))
(input input41 (~5.51 ~5.82) ($36.73 $49.27))
(input input42 (~4.33 ~6.39) ($34.45 $39.55))
(input input43 (~9.03 ~10.29) ($15.84 $22.16))
(input input44 (~6.98 ~9.71) ($68.89 $105.11))
(input input45 (~3.48 ~5.38) ($17.83 $24.17))
(input input46 (~4.62 ~7.32) ($10.58 $15.42))
(input input47 (~3.65 ~4.96) ($9.83 $16.17))
(input input48 (~5.27 ~6.12) ($36.16 $57.84))
(input input49 (~6.29 ~9.67) ($58.54 $71.46))
(input input50 (~2.10 ~3.29) ($52.55 $67.45))
(input input51 (~2.43 ~2.56) ($34.85 $57.15))
(input input52 (~4.15 ~4.56) ($74.41 $81.59))
(input input53 (~5.84 ~9.59) ($64.24 $89.76))
(input input54 (~3.94 ~5.51) ($54.79 $81.21))
(input input55 (~2.67 ~4.33) ($82.16 $111.84))
(input input56 (~2.34 ~3.17) ($47.19 $56.81))
(input input57 (~3.17 ~4.52) ($22.90 $31.10))
(input input58 (~1.62 ~1.68) ($14.98 $17.02))
(input input59 (~4.98 ~6.61) ($41.85 $64.15))
(input input60 (~7.09 ~9.58) ($17.30 $28.70))
(input input61 (~3.47 ~5.19) ($31.46 $44.54))
(input input62 (~2.52 ~4.16) ($12.04 $13.96))
(input input63 (~6.18 ~8.63) ($59.79 $64.21))
(input input64 (~8.29 ~13.30) ($47.93 $54.07))
(input input65 (~8.26 ~10.01) ($74.35 $115.65))
(input input66 (~8.67 ~13.28) ($73.72 $108.28))
(input input67 (~4.94 ~5.13) ($86.34 $109.66))
(input input68 (~4.23 ~6.04) ($51.68 $58.32))
(input input69 (~8.81 ~13.02) ($74.61 $109.39))
(input input70 (~10.81 ~10.89) ($70.42 $103.58))
(input input71 (~7.40 ~7.90) ($36.45 $39.55))
(input input72 (~3.24 ~3.26) ($59.49 $98.51))
(input input73 (~6.74 ~11.22) ($17.23 $26.77))
(input input74 (~6.44 ~8.97) ($51.61 $68.39))
(input input75 (~2.98 ~3.84) ($15.12 $22.88))
(input input76 (~8.63 ~8.67) ($15.25 $18.75))
(input input77 (~8.31 ~10.07) ($33.34 $44.66))
(input input78 (~8.47 ~13.47) ($33.62 $42.38))
(input input79 (~2.00 ~2.55) ($66.38 $71.62))
(input input80 (~8.60 ~10.53) ($95.74 $98.26))
(tree program ~1.12 $0.00
  (comb9 (choice (comb0 (choice input0  input1  input2)
                        (choice input3  input4  input5)
                        (choice input6  input7  input8))
                 (comb1 (choice input9  input10 input11)
                        (choice input12 input13 input14)
                        (choice input15 input16 input17))
                 (comb2 (choice input18 input19 input20)
                        (choice input21 input22 input23)
```

```
                (choice input24 input25 input26)))
(choice (comb3 (choice input27 input28 input29)
               (choice input30 input31 input32)
               (choice input33 input34 input35))
        (comb4 (choice input36 input37 input38)
               (choice input39 input40 input41)
               (choice input42 input43 input44))
        (comb5 (choice input45 input46 input47)
               (choice input48 input49 input50)
               (choice input51 input52 input53)))
(choice (comb6 (choice input54 input55 input56)
               (choice input57 input58 input59)
               (choice input60 input61 input62))
        (comb7 (choice input63 input64 input65)
               (choice input66 input67 input68)
               (choice input69 input70 input71))
        (comb8 (choice input72 input73 input74)
               (choice input75 input76 input77)
               (choice input78 input79 input80)))))
```

THIS PAGE INTENTIONALLY LEFT BLANK

Bibliography

[1] David J. Austin and Patric Jensfelt. Using multiple Gaussian hypotheses to represent probability distributions for mobile robot localization. In *Proc. IEEE International Conference on Robotics & Automation*, pages 1036–1041, 2000.

[2] Omar Benjelloun, Anish Das Sarma, Alon Halevy, Martin Theobald, and Jennifer Widom. Databases with uncertainty and lineage. *The VLDB Journal*, 17(2):243–264, 2008.

[3] David Goldberg. What every computer scientist should know about floating point arithmetic. *ACM Computing Surveys*, 23(1):5–48, 1991.

[4] Noah D. Goodman, Vikash K. Mansinghka, Dan Roy, Keith Bonawitz, and Joshua B. Tenenbaum. Church: A language for generative models. In *Uncertainty in Artificial Intelligence*, 2008.

[5] Pardip Hari, Kevin Ko, Emmanouil Koukoumidis, Ulrich Kremer, Margaret Martonosi, Desiree Ottoni, Li-Shiuan Pei, and Pei Zhang. SARANA: Language, compiler, and run-time system support for spatially aware and resource-aware mobile computing. *Philosophical Transactions of the Royal Society*, 366, October 2008.

[6] Brian Hayes. A lucid interval. *American Scientist*, 91(6):484–488, 2003.

[7] Wendi B. Heinzelman, Amy L. Murphy, Hervaldo S. Carvalho, and Mark A. Perillo. Middleware to support sensor network applications. *IEEE Network*, 18(1):6–14, 2004.

[8] Daphne Koller, David McAllester, and Avi Pfeffer. Effective Bayesian inference for stochastic programs. In *Proceedings of the 14th National Conference on Artificial Intelligence (AAAI)*, pages 740–747. AAAI Press, 1997.

[9] Samuel Ross Madden. *The design and evaluation of a query processing architecture for sensor networks*. PhD thesis, University of California at Berkeley, Berkeley, CA, USA, 2003.

[10] Silvano Martello and Paolo Toth. *Knapsack Problems: Algorithms and Computer Implementations*. John Wiley & Sons, Chichester, England, 1990.

[11] Sungwoo Park, Frank Pfenning, and Sebastian Thrun. A probabilistic language based upon sampling functions. *SIGPLAN Not.*, 40(1):171–182, 2005.

[12] Avi Pfeffer. IBAL: A probabilistic rational programming language. In *IJCAI-01*, pages 733–740. Morgan Kaufmann Publishers, 2001.

[13] Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, Cambridge, MA, USA, 2002.

[14] Alexey Radul. Report on the probabilistic language Scheme. In *DLS '07: Proceedings of the 2007 Symposium on Dynamic Languages*, pages 2–10, New York, NY, USA, 2007. ACM.

[15] Norman Ramsey and Avi Pfeffer. Stochastic lambda calculus and monads of probability distributions. In *29th ACM POPL*, pages 154–165. ACM Press, 2002.

[16] Armando Solar-Lezama, Liviu Tancau, Rastislav Bodik, Vijay Saraswat, and Sanjit Seshia. Combinatorial sketching for finite programs. In *ASPLOS-XII: Proceedings of the 12th international conference on Architectural Support for Programming Languages and Operating Systems*, pages 404–415, New York, NY, USA, 2006. ACM.

[17] Ryo Sugihara and Andrew A. Chien. Accuracy-aware data modeling in sensor networks. In *SenSys '05: Proceedings of the 3rd International Conference on Embedded Networked Sensor Systems*, pages 282–283, New York, NY, USA, 2005. ACM.

[18] Ryo Sugihara and Andrew A. Chien. An efficient general-purpose mechanism for data gathering with accuracy requirement in wireless sensor networks. Technical Report CS2006-0911, University of California, San Diego, October 2007.

[19] Ryo Sugihara and Rajesh K. Gupta. Programming models for sensor networks: A survey. *ACM Transactions on Sensor Networks*, 4(2):1–29, 2008.

[20] Sebastian Thrun. Probabilistic algorithms in robotics. *AI Magazine*, 21(4):93–109, 2000.

[21] Sebastian Thrun. A programming language extension for probabilistic robot programming. In *Workshop notes of the IJCAI Workshop on Uncertainty in Robotics (RUR)*. IJCAI, Inc, 2000.

[22] Sebastian Thrun, Wolfram Burgard, and Dieter Fox. *Probabilistic Robotics*. MIT Press, Cambridge, MA, USA, 2005.